Programmation Avancée et Application

Techniques avancées en Java

Jean-Guy Mailly

jean-guy.mailly@u-paris.fr

LIPADE - Université de Paris
http://www.math-info.univ-paris5.fr/∽jmailly/

Techniques avancées en Java

1. Types génériques
2. Énumérations
3. Exceptions
4. Entrées sorties
5. Introduction à la modularité
6. Classes internes et lambda expressions

Types génériques

Le cas des listes

- En Java « à l'ancienne », on faisait ArrayList l = **new** ArrayList();
- Problème : quand on accède à un élément de la liste, on ne peut pas le manipuler sans le caster de la bonne manière. Par exemple, même si on sait que l ne contient que des String, on ne peut pas faire String s = l.get(0); ni

```
Object o = l.get(0);
System.out.println(o.length());
```

- Solution avec un cast :

```
String s = (String)l.get(0);
System.out.println(s.length());
```

Le cas des listes

- En Java « à l'ancienne », on faisait ArrayList l = **new** ArrayList();
- Problème : quand on accède à un élément de la liste, on ne peut pas le manipuler sans le caster de la bonne manière. Par exemple, même si on sait que l ne contient que des String, on ne peut pas faire String s = l.get(0); ni

```
Object o = l.get(0);
System.out.println(o.length());
```

- Solution avec un cast :

```
String s = (String)l.get(0);
System.out.println(s.length());
```

⚠Plusieurs problèmes :

- Code peu lisible s'il y a trop de cast
- Risque d'erreur important car on n'est pas sûr à la compilation du type des données manipulées

Les listes génériques

- Dans la définition des listes, on peut utiliser un paramètre qui décrit le types des éléments de cette liste
- Avantages :
  - Vérification des types dès la compilation : détection et correction d'erreurs plus facile
  - Moins de cast ($\rightarrow$ code plus léger)
  - Possibilité d'écrire des algorithmes génériques, c'est-à-dire des algorithmes qui se comportent globalement de la même manière quel que soit le type des données
    $\rightarrow$ pour trier des nombres dans l'ordre croissant, ou trier des chaînes de caractères dans l'ordre alphabétique, l'algorithme est globalement le même

Définition d'une classe génériques

- On peut paramétrer la définition d'une classe générique par plusieurs types :

```
public class MaClasse<T1,T2,...,Tn>{
  // ...
}
```

- Conventions pour les noms :
  - E : pour des types d'éléments (cf List<E>)
  - K et V : respectivement pour les clés et valeurs (cf Map<K,V>)
  - N : pour des types numériques
  - Pour tout autre type, T (puis S pour le deuxième type, U pour le troisième, V pour le quatrième,...)
- On peut également créer des interfaces génériques (par exemple, List<E> est une interface, tandis que ArrayList<E> est une classe qui implémente List<E>)

Exemple de classe générique simple

```
public class Couple<T, U> {
  private T premier ;
  private U second ;

  public Couple(T p, U s) {
    premier = p ;
    second = s ;
  }

  public T getPremier() {
    return premier ;
  }

  public U getSecond() {
    return second ;
  }
}
```

Instances concrètes de la classe générique

- On peut créer des objets qui appartiennent à différents types de couples en fonction des valeurs données aux paramètres T et U
- Les valeurs peuvent être identiques ou différentes, et peuvent même appartenir à un type paramétré
- Exemples :
  - Couple<String,String> c1 = **new** Couple<>("toto","titi");
  - Couple<Double,Integer> c2 = **new** Couple<>(2.3,4);
  - Couple<String,Double> c3 = **new** Couple<>("toto",1.0);
  - Couple<Double,Double> c4 = **new** Couple<>(1.0,3.5);
  - Couple<Couple<Double,Double>,Couple<Double,Double>> c5 = **new** Couple<>(c4,**new** Couple<>(2.3,5.1));

La généricité cachée (1/3)

- On peut créer des classes qui héritent d'une classe générique afin de simplifier certains traitements, et bénéficier des méthodes définies dans la classe générique

```
public class Point extends Couple<Double, Double> {
  public Point(Double abs, Double ord) {
    super(abs, ord);
  }

  public double getAbscisse() {
    return getPremier();
  }

  public double getOrdonnee() {
    return getSecond();
  }
}
```

La généricité cachée (2/3)

```
public class Personne extends Couple<String, String> {
  public Personne(String prenom, String nom) {
    super(prenom, nom);
  }

  public String getPrenom() {
    return getPremier();
  }

  public String getNom() {
    return getSecond();
  }
}
```

La généricité cachée (3/3)

- On suppose qu'on a défini une classe Etudiant, et on définit une promotion comme étant une liste d'étudiants

```
public class Promotion extends ArrayList<Etudiant> {
  // ...
}
```

- On peut utiliser directement toutes les méthodes d'une ArrayList (isEmpty(), add(),...), et ajouter des méthodes propres à une promotion d'étudiants (par exemple calculMoyenneDePromo())

Inférence de types

- Il n'est pas nécessaire de préciser les types lors de l'utilisation de **new** si le compilateur peut les déduire du contexte. Exemples :
  - ArrayList<Integer> l = **new** ArrayList<>();
  - Couple<String,String> c = **new** Couple<>();
- Ne pas oublier les chevrons <>, sinon cela définit un type brut (*raw type*). Les types bruts sont conservés pour la compatibilité du vieux code (avant l'introduction des types génériques), mais il n'est pas recommandé de les utiliser

Méthodes génériques

- Une méthode générique est une méthode qui introduit un (ou des) type(s) paramétré(s).
- Une méthode générique peut être définie dans une classe générique, ou dans une classe non générique
- Les paramètres sont introduits entre chevrons avant le type de retour de la méthode

```
public class Util {
  public static <T, U> boolean compareCouples(
                Couple<T, U> c1, Couple<T, U> c2) {
    return c1.getPremier().equals(c2.getPremier())
          && c1.getSecond().equals(c2.getSecond());
  }
}
```

Appel d'une méthodes génériques

- Syntaxe complète :

```
Couple<String,Double> c1 = new Couple<>("toto",2.3);
Couple<String,Double> c2 = new Couple<>("titi",2.2);
boolean b = Util.<String,Double>compareCouples(c1,c2);
```

Appel d'une méthodes génériques

- Syntaxe complète :

```
Couple<String,Double> c1 = new Couple<>("toto",2.3);
Couple<String,Double> c2 = new Couple<>("titi",2.2);
boolean b = Util.<String,Double>compareCouples(c1,c2);
```

- Syntaxe allégée grâce à l'inférence de types du compilateur :

```
Couple<String,Double> c1 = new Couple<>("toto",2.3);
Couple<String,Double> c2 = new Couple<>("titi",2.2);
boolean b = Util.compareCouples(c1,c2);
```

Types bornés : intuition

- Supposons qu'on veut écrire un algorithme qui dépend de la méthode f() qui est définie dans le classe (ou l'interface) A
- Différentes classes héritent de la classe A (ou implémentent) l'interface A, et on ne sait pas à l'avance quels types seront utilisés lors de l'exécution
- On peut indiquer dans la signature d'une méthode qu'on a besoin d'utiliser un type paramétré qui est un sous-type de A
- On va montrer sur un exemple qu'on ne peut pas toujours utiliser directement A quand on a besoin de ses sous-types

Types bornés : intuition

```
public void algo(List<A> l){
  for(A a : l){
    a.f() ;
  }
}
```

- Soit A1 une classe qui hérite de A, et l une liste d'éléments de type A1. L'instruction algo(l) ; provoque l'erreur suivante :

```
The method algo(List<A>) in the type Util is not applicable
for the arguments (List<A1>)
```

- **Remarque** : Même quand A1 est un sous-type de A, MaClasse<A1> et MaClasse<A> ne sont pas parentes

Types bornés : borne supérieure

- On peut indiquer que les éléments de l doivent appartenir à un sous-type de A
- On utilise le mot-clé **extends**, que A soit une classe ou une interface

```
public <T extends A> void algo(List<T> l){
  for(T a : l){
    a.f() ;
  }
}
```

- On peut maintenant appeler algo(l) ; avec l de type List<A>, List<A1>, ou n'importe quel autre type List<XXXX>, avec XXXX un sous-type de A

Généricité et héritage

- *Même quand A1 est un sous-type de A, MaClasse<A1> et MaClasse<A> ne sont pas parentes* : explication sur un exemple concret

```
List<Integer> l1 = new ArrayList<Integer>() ;
List<Number> l2 = l1 ;
```

✘Erreur :

```
Type mismatch: cannot convert from List<Integer> to
List<Number>
```

- Comme l2 est de type List<Number>, on devrait pouvoir ajouter dedans n'importe quel objet de type Number
- Problème : concrètement, l2 est une référence vers un objet de type ArrayList<Integer>, on ne peut donc pas ajouter n'importe quel Number dedans

Types bornés : bornes multiples

- On peut spécifier plusieurs bornes supérieures :
  <T **extends** A & B & C>
- Les bornes supérieures peuvent être (éventuellement) une classe-mère, et des interfaces
- Si une des bornes est une classe-mère, elle doit être donnée en première position de la liste :

```
public class A { ... }
public interface B { ... }
public interface C { ... }

public <T extends B & C & A> f(){ ... }
```

✘Ne fonctionne pas car la classe A n'est pas donnée en premier

Types bornés : bornes multiples

- On peut spécifier plusieurs bornes supérieures :
  <T **extends** A & B & C>
- Les bornes supérieures peuvent être (éventuellement) une classe-mère, et des interfaces
- Si une des bornes est une classe-mère, elle doit être donnée en première position de la liste :

```
public class A { ... }
public interface B { ... }
public interface C { ... }

public <T extends A & B & C> f(){ ... }
```

✔Ok : la classe A est donnée en premier

Jokers

- Le point d'interrogation est un joker (en anglais, *wildcard*) pour représenter un type inconnu
- On peut définir une méthode générique qui fonctionne avec n'importe quel sous-type d'un type A, sans avoir besoin de déclarer un type en paramètre :

```
public void f(List<? extends Number> l){
  for(Number n : l){
    // ...
  }
}
```

- Cette méthode fonctionne avec différents types de paramètres List<Number>, List<Integer>, List<Double>,...

Jokers : borne inférieure

- On peut indiquer avec un joker qu'on a besoin d'objets qui appartiennent à un type parent d'un d'autre type
- Exemple : pour tester si tous les éléments d'une liste d'Integer apparaissent dans une certaine collection, on peut limiter les objets de la collection à un type parent d'Integer (Number or Object) :

```
public static boolean containsAll(List<Integer> l,
                                  Collection<? super Integer> c) {
  for (Integer i : l) {
    if (!c.contains(i))
      return false;
  }
  return true;
}
```

Énumérations

Énumérations

- Une énumération est une classe particulière qui a un nombre fini et prédéfini d'instances
- Utilisation basique d'une énumération : l'exemple du jeu de cartes

```
public enum CouleurCarte {
  COEUR, CARREAU, PIQUE, TREFLE ;
}
```

- L'énumération est juste une liste de constantes, séparées par des virgules, qui se termine par un point virgule
- Utilisation d'une constante de l'enum :
  CouleurCarte c = CouleurCarte.COEUR ;

Énumérations et switch

- Les types énumérés peuvent être utilisés dans une instruction **switch-case**

```
CouleurCarte c = CouleurCarte.COEUR;

switch (c) {
  case COEUR:
    System.out.println("C'est du coeur."); break;
  case CARREAU:
    System.out.println("C'est du carreau."); break;
  case PIQUE:
    System.out.println("C'est du pique."); break;
  case TREFLE:
    System.out.println("C'est du trefle."); break;
}
```

Énumérations plus élaborées : personnages de jeu

```
public enum TypePerso {
  MAGE(4,7,3), GUERRIER(7,0,7),
  ELFE(5,2,9), NAIN(6,0,6) ;

  private final int force ;
  private final int magie ;
  private final int mouvement ;

  private TypePerso(int force,int magie,int mouvement){
    this.force = force ;
    this.magie = magie ;
    this.mouvement = mouvement ;
  }
  public int getForce() { return force; }
  public int getMagie() { return magie; }
  public int getMouvement() { return mouvement; }
}
```

Méthodes utiles

- name() : renvoie un String correspondant au nom de la constante

```
public void f(CouleurCarte c){
   System.out.println("C'est␣du␣" + c.name() + ".");
}
```

- valueOf() : renvoie la constance correspondant à un String

```
CouleurCarte c = CouleurCarte.valueOf("COEUR") ;
```

- values() : renvoie un tableau composé des constantes de l'énumération dans l'ordre de leur déclaration
- ordinal() : renvoie l'indice d'une constante dans le tableau retourné par la méthode values()

```
CouleurCarte[] tab = CouleurCarte.values();
// [COEUR, CARREAU, PIQUE, TREFLE]
System.out.println(CouleurCarte.CARREAU.ordinal());
// Affiche 1
```

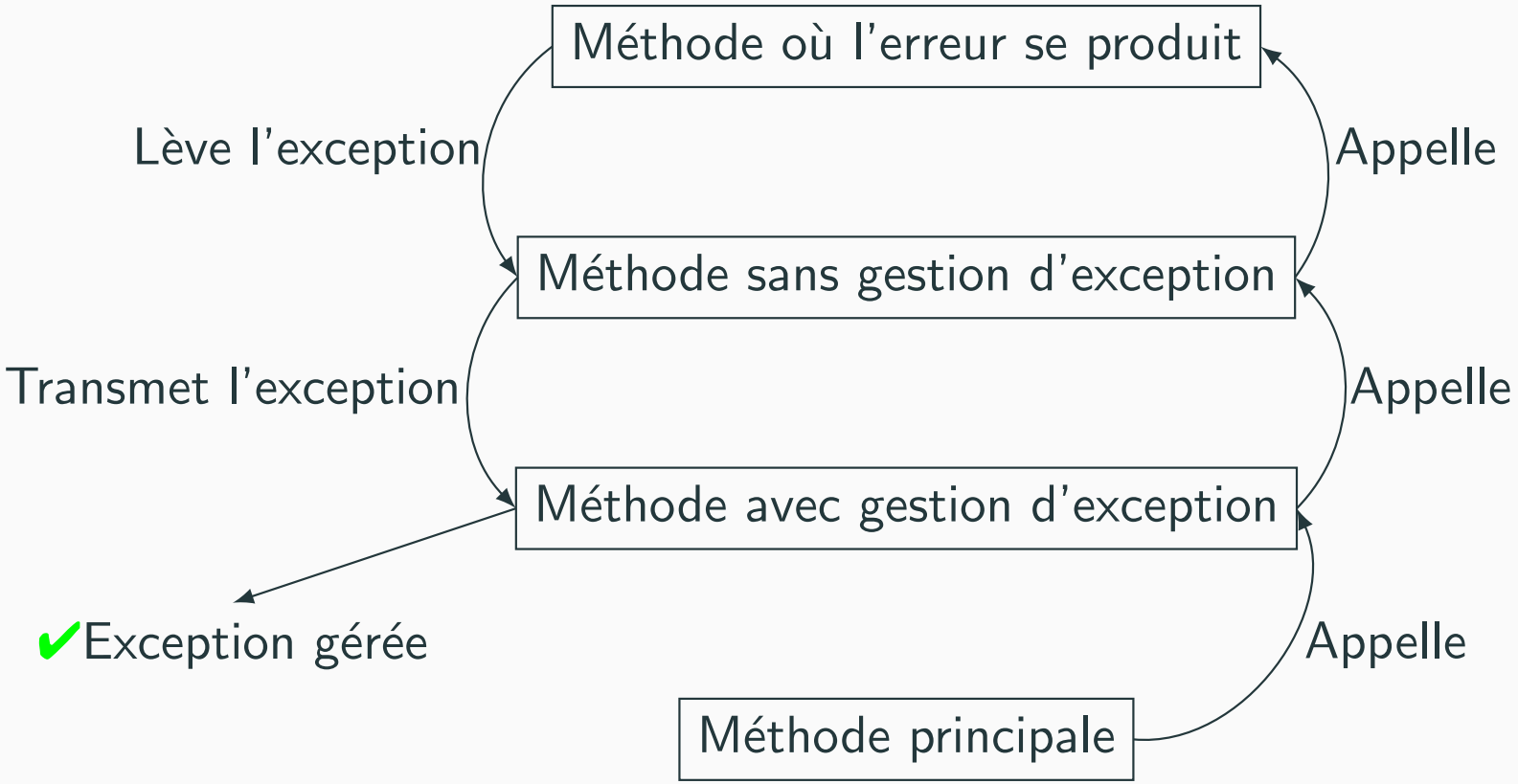
Exceptions

Qu'est-ce qu'une exception ?

- Une exception dans un programme Java est un comportement inhabituel qui provoque une erreur d'exécution
- Elle peut être gérée pour reprendre la bonne marche du programme, ou éventuellement permettre un arrêt « propre » du programme (par exemple, en fermant les flux ouverts)
- Si une exception n'est pas gérée, elle provoque un arrêt brutal du programme
- Lorsqu'une erreur survient, un objet est créée, qui contient un certain nombre d'informations sur l'erreur : on dit que l'exception est *levée*

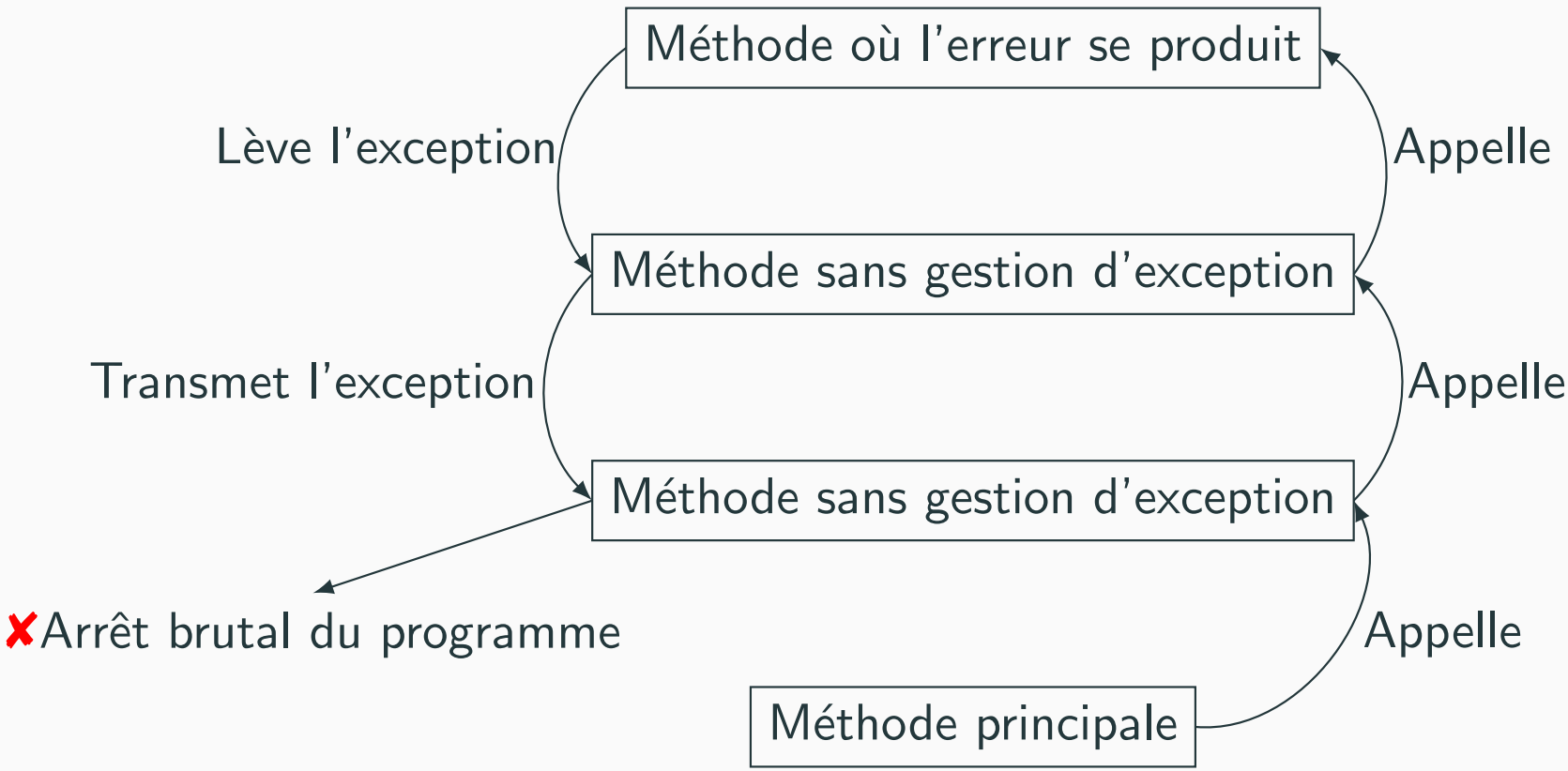
Gestion d'exception : la pile d'exécution

- L'exception doit être gérée dans une des méthodes de la pile d'exécution qui a mené à l'exception



Gestion d'exception : la pile d'exécution

- L'exception doit être gérée dans une des méthodes de la pile d'exécution qui a mené à l'exception



La classe Exception

- Exception est la classe mère de toutes les exceptions. De nombreuses classes filles sont définies
  - IOException : erreurs lors d'entrées/sorties du programme
  - FileNotFoundException : si on essaye d'accéder à un fichier qui n'existe pas (classe fille de IOException)
  - ArrayIndexOutOfBoundsException : si on essaye d'accéder à un indice d'un tableau qui n'existe pas
  - NullPointerException : si on essaye d'utiliser un objet qui vaut **null**
  - NumberFormatException : si on essaye de convertir une String en nombre de manière incorrecte
  - ...
- On peut créer ses propres exceptions en créant une nouvelle classe fille d'Exception ; par convention, les noms des classes d'exceptions se terminent tous par Exception

Lever une exception (1/4)

```
public static long factorielle(long n) {
  if (n == 1)
    return n;
  else
    return n * factorielle(n - 1);
}
```

- Problème : on ne peut pas calculer la factorielle d'un nombre ≤ 0
- Solution : lever une exception si l'utilisateur essaye de le faire

Lever une exception (2/4)

```
public static long factorielle(long n)
                              throws Exception {
  if (n < 1)
    throw new Exception();
  if (n == 1)
    return n;
  else
    return n * factorielle(n - 1);
}
```

- Le mot-clé **throw** suivi d'un objet de type Exception permet de lever l'exception
- Le mot-clé **throws** dans la signature indique que la méthode est susceptible de lever une exception

Lever une exception (3/4)

```
public static long factorielle(long n)
                throws Exception {
  if (n < 1)
    throw new Exception (
                "On ne peut pas calculer la "
      + "factorielle d'un nombre negatif : " + n);
  if (n == 1)
    return n;
  else
    return n * factorielle(n - 1);
}
```

- Il est recommandé de transmettre un message qui décrit précisément l'erreur dans le constructeur de l'exception

Lever une exception (4/4)

```
public static long factorielle(long n)
                throws MauvaisArgumentException {
  if (n < 1)
    throw new MauvaisArgumentException(
                        "On ne peut pas calculer la "
      + "factorielle d'un nombre negatif : " + n);
  if (n == 1)
    return n;
  else
    return n * factorielle(n - 1);
}
```

- Il vaut mieux utiliser le type d'exception le plus spécifique possible pour le problème rencontré, quitte à créer ses propres classes d'exceptions

Gérer une exception (1/2)

```
public static void calculerDesFactorielles(int n) {
  for(int i = 1 ; i <= n ; i++) {
    factorielle(i);
  }
}
```

✘ `Unhandled exception type MauvaisArgumentException`

Gérer une exception (2/2)

```
public static void calculerDesFactorielles(int n) {
  for(int i = 1 ; i <= n ; i++) {
    try {
      factorielle(i);
    } catch (MauvaisArgumentException e){
      // Gestion de l'exception
    }
  }
}
```

- La partie gestion de l'exception peut servir à afficher une information utile à l'utilisateur avant de quitter le programme proprement, ou au contraire à corriger l'erreur

Rappel : la calculatrice

```
private static void boucle(Scanner sc) {
  int choix;
  do {
    menu();
    choix = sc.nextInt();
    switch (choix) {
      case 0: break;
      case 1: addition(sc); break;
      case 2: soustraction(sc); break;
      default:
        System.out.println(
        "Ce choix n'est pas conforme.");
    }
  } while (choix != 0);
}
```

La méthode nextInt()

- La Javadoc indique la méthode nextInt est susceptible de lever plusieurs exceptions, dont :
  InputMismatchException : if the next token does not match the Integer regular expression, or is out of range
- On peut donc modifier la méthode boucle(Scanner sc) pour s'assurer que l'utilisateur entre bien un nombre entier au clavier

Gérer les exceptions de la calculatrice

- On doit remplacer la ligne choix = sc.nextInt(); :

```
boolean choixEstFait = false;
do {
   try {
      choix = sc.nextInt();
      choixEstFait = true ;
   } catch (InputMismatchException e) {
      System.out.println("Choix␣incorrect");
      sc.nextLine();
      menu();
   }
} while (!choixEstFait);
```

- Si sc.nextInt() lève une exception, la valeur du booléen n'est pas modifiée, donc on ne sort pas de la boucle

Gérer les exceptions de la calculatrice

Pour vous entraîner, gérez les exceptions dans les autres méthodes de la calculatrice (addition/soustraction)

Transmettre une exception

- Il est possible de ne pas gérer une exception dans la méthode où elle se produit : on la laisse passer vers la méthode appelante
- L'exception doit alors être gérée par la méthode appelante, ou alors elle doit être à nouveau explicitement laissée passer

```
public static void calculerDesFactorielles(int n)
                        throws MauvaisArgumentException{
  for(int i = 1 ; i <= n ; i++) {
    factorielle(i);
  }
}
```

- Rappel : si on ne fait que laisser passer l'exception sans jamais la gérer, le programme se termine brutalement lorsque l'exception atteint la méthode main

RuntimeException

- RuntimeException est la classe mère des exceptions qui peuvent être levées par le comportement normal de la JVM
- Cette classe (et ses filles) ont la particularité d'être des exceptions *unchecked* : il n'est pas obligatoire de les gérer avec un **try-catch** ou de les signaler avec le mot-clé **throws**
- Cela se justifie par le fait que leur levée n'est pas facile à prédire
- Exemples :
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - IllegalArgumentException
  - NullPointerException
  - ...

Multiples catch

- Un même bloc **try** peut être suivi de plusieurs blocs **catch** si plusieurs exceptions peuvent être levées par le même code

```
try {
  // ...
} catch ( ... ){
  // ...
} catch ( ... ){
  // ...
}
```

- Par exemple, une exception peut être levée par le même bloc d'instructions à cause de :
  - l'ouverture d'un fichier
  - le format des données lues dans le fichier
  - on essaye de lire alors que le fichier est terminé
  - ...

Multiples catch et héritage

- Si un même bloc peut lever des exceptions de type A et B, avec A **extends** B, il faut faire attention à l'ordre des **catch**

```
try {
  // ...
} catch(B b){
  // Gestion speciale de l'exception B
} catch(A a){
  // Gestion speciale de l'exception A
}
```

- Si une exception de type A arrive, comme A hérite de B, elle va être gérée par le premier **catch**

Multiples catch et héritage

- Si un même bloc peut lever des exceptions de type A et B, avec A **extends** B, il faut faire attention à l'ordre des **catch**

```
try {
  // ...
} catch(A a){
  // Gestion speciale de l'exception A
} catch(B b){
  // Gestion speciale de l'exception B
}
```

- Les exceptions de type A sont gérées par le **catch** dédié

Finally

- Le mot-clé **finally** permet de créer un bloc d'instructions qui s'exécute quoi qu'il arrive après le **try-catch**
- Le bloc **finally** permet d'exécuter du code même si une exception imprévue se produit
  - Utile pour le code de « nettoyage », par exemple pour refermer un flux d'entrées/sorties

```
Scanner sc = new Scanner(System.in);
try {
  System.out.println(sc.nextInt());
} catch( ... ) {
  // ...
} finally {
  sc.close() ;
}
```

Entrées sorties

Entrées sorties

- Rappel : on a vu précédemment les entrées-sorties de base (System.out et System.in + Scanner)
- Nous allons voir ici quelques approches plus élaborées, notamment pour la manipulation de fichiers
- En plus des classes et méthodes que nous verrons dans ce cours, d'autres outils sont fournis par le package java.io

InputStream : la base de la lecture de données

- Classe abstraite, mère de toutes les classes qui correspondent à des flux d'entrée d'octets
- La méthode abstrait read() doit être définie par les classes filles concrètes ; elle retourne un **int** qui correspond au prochain octet à lire dans le flux d'entrée
- Entrée de bas niveau : on récupère directement les octets sans tenir compte du genre de données manipulé
- On verra les classes filles FileInputStream et ObjectInputStream

OutputStream : la base de l'écriture de données

- Contrepartie d'InputStream : classe abstraite, mère de toutes les classes qui correspondent à des flux de sortie d'octets
- La méthode abstrait write(**int** b) doit être définie par les classes filles concrètes ; elle écrit un octet donnée sous forme d'**int** dans le flux de sortie
- Sortie de bas niveau : on récupère directement les octets sans tenir compte du genre de données manipulé
- On verra notamment les classes filles FileOutputStream et ObjectOutputStream

PrintStream : ajout de fonctionnalités à un OutputStream

- Les différentes version de la méthode print (...) permettent d'écrire des données de différents types dans flux de sortie

```
OutputStream os = ... ;
// n'importe quel type d'OutputStream
PrintStream ps = new PrintStream(os) ;
ps.print('c');
ps.print(3.2);
ps.print("toto");
ps.print(new MaClasse());
```

- Les versions de println (...) permettent la même chose, avec l'ajout d'un retour à la ligne après l'écriture des données passées en paramètre
- L'objet System.out est une instance de PrintStream

Manipulation de fichiers : la classe File

- File permet de manipuler des fichiers et des répertoires grâce au chemin qui les identifie
- L'attribut separator permet de construire un chemin en utilisant le séparateur adapté au système d'exploitation (/ sous Unix, \\ sous Windows)
- Exemple : on veut ouvrir le fichier `toto.txt`, situé dans le répertoire `rep`, lui-même situé dans le répertoire `repParent` :

```
File f = new File("repParent" + File.separator
                  + "rep" + File.separator + "toto.txt");
```

Quelques méthodes de la classe File

- canRead/canWrite/canExecute : testent si le programme a les droits de lecture/écriture/exécution sur le fichier
- createNewFile()/mkdir() : crée un fichier/répertoire s'il n'existe pas déjà
- delete () : supprime un fichier ou répertoire
- exists () : teste si le fichier existe
- getParent() : retourne le chemin vers le parent du fichier sous forme de String
- isDirectory ()/ isFile () : testent si le fichier est un répertoire ou un fichier ordinaire
- ...

De nombreuses autres méthodes sont décrites dans la Javadoc :
java . io . File

Lire et écrire des données brutes dans un fichier

- FileInputStream et FileOutputStream : lecture/écriture de données brutes
- Instances construites avec un File en paramètre, ou un String
- La lecture et l'écriture se font avec les méthodes de bas niveau héritées des classes InputStream et OutputStream
- Utile si on doit manipuler directement des données non textuelles (image, son, fichier exécutable,...)

Sérialisation

- La sérialisation est l'écriture de données représentées sous forme d'objets dans un flux de sortie :
  - dans un fichier pour la persistance des données
  - dans une socket réseau pour la communication entre plusieurs machines
  - ...
- Utilisation de la classe ObjectOutputStream

```
FileOutputStream fos = new FileOutputStream(
                                        "toto.txt");
ObjectOutputStream oos = new ObjectOutputStream(
                                            fos);
oos.writeObject(new MaClasse());
```

- Si on souhaite sérialiser une instance d'une classe A, tous les attributs de A doivent être eux-mêmes sérialisables, sauf s'ils sont statiques ou identifiés par le mot-clé **transient**

Désérialisation

- Le processus inverse (→ décodage des données depuis un flux pour reformer un objet Java) est appelé désérialisation
- Utilisation de la classe ObjectInputStream
- La désérialisation avec readObject() de la classe ObjectInputStream retourne un Object, il faut donc faire un transtypage vers le type souhaité

```
FileInputStream fis = new FileInputStream(
                                        "toto.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
MaClasse mc = (MaClasse) ois.readObject();
```

- Si on désérialise un objet dont un attribut est **transient**, cet attribut est initialisé à **null**

L'interface Serializable

- On ne peut pas sérialiser un objet si la classe à laquelle il appartient n'implémente pas l'interface Serializable
- Cette interface n'a pas d'attributs ni de méthodes, elle sert uniquement à indiquer qu'un objet peut être sérialisé/désérialisé avec les méthodes writeObject et readObject des classes ObjectOutputStream et ObjectInputStream

Reader/Writer

- Classes abstraites, respectivement mères des classes dédiées à la lecture et à l'écriture de flux de caractères
- Pour des données textuelles, il n'y a pas besoin de gérer soi-même l'encodage des caractères en octets
- read(**char**[] buf) de la classe Reader lit des caractères jusqu'à la fin du flux d'entrée (ou l'arrivée d'une erreur), et les stocke dans le tableau buf passé en paramètre
- La classe Writer fournit (entre autres) les méthodes write(**char**[] buf) et write(String s) pour écrire des caractères (ou une chaîne de caractères) dans un flux

Faire le lien entre IO de bas niveau et IO de haut niveau

- InputStreamReader est une classe fille de Reader qui permet de passer d'un flux d'octets (→ InputStream) à un flux de caractères
- La conversion est faite grâce à la connaissance de l'encodage (*charset*) utilisé, qui peut être donné en paramètre (ou par défaut, celui du système)
- De manière symétrique, OutputStreamWriter est une classe fille de Writer qui permet de convertir des caractères en octets pour les transmettre à un OutputStream

Lire et écrire dans un fichier

- FileReader et FileWriter sont des classes filles (respectivement) d'InputStreamReader et OutputStreamWriter, qui sont dédiées à la lecture et l'écriture dans des fichiers
- Contrairement aux FileInputStream et FileOutputStream, ces classes sont adaptées à la lecture/écriture de données textuelles ($\rightarrow$ caractères)

Lecture bufferisée (1/2)

- BufferedReader est une classe fille de Reader qui propose une lecture efficace de caractères et de lignes dans un flux d'entrée
- un BufferedReader peut uniquement lire dans un autre Reader, par exemple :
  - Dans un InputStreamReader pour faire le lien avec un InputStream quelconque

```
InputStream is = ... ;
InputStreamReader isr = new InputStreamReader(is);
BufferedReader bReader = new BufferedReader(isr);
```

  - Dans un FileReader

```
FileReader fReader = new FileReader("toto.txt");
BufferedReader bReader = new BufferedReader(fReader);
```

Lecture bufferisée (2/2)

- readLine() est une méthode de BufferedReader qui lit une ligne à partir du flux d'entrée et la retourne sous forme de String
- readLine() retourne **null** lorsque la fin du fichier est atteinte
- Un exemple de procédure standard pour lire les lignes d'un fichier est donc :

```
FileReader fReader = new FileReader("toto.txt");
BufferedReader bReader=new BufferedReader(fReader);
String ligne = null;

while((ligne = bReader.readLine()) != null){
  // faire quelque chose avec la String obtenue
  ...
}
```

Écriture bufferisée

- De manière symétrique, BufferedWriter est une classe fille de Writer qui propose une écritures efficace de caractères dans un flux de sortie
- un BufferedWriter peut uniquement écrire dans un autre Writer, et rend les écritures plus efficaces :

```
BufferedWriter bW = new BufferedWriter(
                        new FileWriter("toto.txt")) ;
PrintWriter pW = new PrintWriter(bW) ;
```

- La classe PrintWriter fournit des méthodes d'écriture similaires à celles de PrintStream (print (...) / println (...))
- Sans le BufferedWriter, l'appel à pw. println (...) nécessiterait, pour chaque caractère à écrire, sa conversion en octet et son écriture sur le fichier → nombreux accès au disque dur
- Avec le BufferedWriter, les caractères sont convertis en octets et écrits sur le fichier par « paquets » → moins d'accès au disque dur

Gestion des erreurs : IOException

- Les exemples vus jusque là ignorent les erreurs qui peuvent arriver lors des entrées et sorties
- De nombreuses méthodes de lecture/écriture dans des flux peuvent provoquer des erreurs
- IOException est la classe mère des exceptions d'entrées/sorties :
  - FileNotFoundException : si le fichier qu'on essaye d'ouvrir en lecture ou écriture n'existe pas ou est inaccessible (p.ex. si on essaye d'écrire dans un fichier qui est en lecture seule)
  - SocketException : indique une erreur lors de la création ou l'accès à une socket réseau
  - EOFException : indique que la fin du flux a été atteinte de manière inattendue
  - ...
- Il est bien entendu **vivement recommandé** de gérer ses exceptions à l'aide d'un **try-catch**

Try avec ressources et fermeture des flux

- Depuis Java 7, un bloc **try** peut être modifié pour gérer les ressources, c'est-à-dire un objet qui doit être fermé lorsqu'on a fini de l'utiliser, comme c'est le cas des InputStream, OutputStream, Reader et Writer : utilisation de la méthode `close()`

Try avec ressources et fermeture des flux

- Depuis Java 7, un bloc **try** peut être modifié pour gérer les ressources, c'est-à-dire un objet qui doit être fermé lorsqu'on a fini de l'utiliser, comme c'est le cas des InputStream, OutputStream, Reader et Writer : utilisation de la méthode close()

Avant Java 7 :

```
public void lire(String nomFichier) throws IOException{
  BufferedReader br = new BufferedReader(
                      new FileReader(nomFichier));
  try {
    System.out.println(br.readLine());
  } finally {
    if(br != null){
      br.close() ;
    }
  }
}
```

Try avec ressources et fermeture des flux

- Depuis Java 7, un bloc **try** peut être modifié pour gérer les ressources, c'est-à-dire un objet qui doit être fermé lorsqu'on a fini de l'utiliser, comme c'est le cas des InputStream, OutputStream, Reader et Writer : utilisation de la méthode close()

Depuis Java 7 :

```
public void lire(String nomFichier) throws IOException{
  try (BufferedReader br = new BufferedReader(
                    new FileReader(nomFichier))) {
    System.out.println(br.readLine());
  }
}
```

- La méthode close() sera appelée implicitement, que l'exception soit levée ou pas

Try avec plusieurs ressources

- Plusieurs ressources peuvent être utilisées par le même bloc **try** :

```
try (Ressource1 r1 = new Ressource1() ;
     Ressource2 r2 = new Ressource2() ;
     ...
     RessourceN rN = new RessourceN()){
  // Instructions du bloc try
}
```

- Les méthodes close() des différentes ressources seront (implicitement) appelées dans l'ordre inverse de leur création (d'abord rN.close(),... pour finir par r2.close() et r1.close())

Exemple de lecture d'un fichier : la classe Personne

```
public class Personne {
  private String nom;
  private String prenom;
  private String telephone;

  public Personne(String nom, String prenom,
                              String telephone) {
    this.nom = nom;
    this.prenom = prenom;
    this.telephone = telephone;
  }
  public String toString() {
    return "(" + nom + "," + prenom + ","
                           + telephone + ")";
  }
  // ...
}
```

Fichier de sauvegarde d'une Personne

On peut stocker une personne dans un fichier qui respecte le format suivant :

```
nom:XXX
prenom:YYY
telephone:ZZZ
```

où `XXX`, `YYY` et `ZZZ` sont les valeurs (respectivement) du nom, du prénom et du numéro de téléphone
Il suffit d'écrire une classe qui lit un fichier contenant ces données pour initialiser une Personne dans l'application à partir de fichiers (qui peuvent avoir été générés par une autre application, ou directement écrit « à la main »)

Exemple de lecture d'un fichier : le parser (1/6)

```
public class PersonneParser {
  public static Personne parser(String fichier) {
    String nom = null;
    String prenom = null;
    String telephone = null;

    try (BufferedReader br = new BufferedReader(
                             new FileReader(fichier))) {
      String ligne = null;
```

Exemple de lecture d'un fichier : le parser (2/6)

```
while ((ligne = br.readLine()) != null) {
  if (ligne.startsWith("nom")) {
    if (nom != null) {
      System.err.println("Il ne peut pas y avoir"
                         + " deux lignes pour le nom");
      System.exit(0);
    } else {
      nom = ligne.split(":")[1];
    }
  }
```

Exemple de lecture d'un fichier : le parser (3/6)

```
else if (ligne.startsWith("prenom")) {
  if (prenom != null) {
    System.err.println("Il␣ne␣peut␣pas␣y␣avoir"
                       + "␣deux␣lignes␣pour␣le␣prenom");
    System.exit(0);
  } else {
    prenom = ligne.split(":")[1];
  }
}
```

Exemple de lecture d'un fichier : le parser (4/6)

```
    else if (ligne.startsWith("telephone")) {
      if (telephone != null) {
        System.err.println("Il ne peut pas y avoir"
                + "deux lignes pour le telephone");
        System.exit(0);
      } else {
        telephone = ligne.split(":")[1];
      }
    } else {
      System.err.println("Cette ligne est"
                         + " incorrecte :");
      System.err.println(ligne);
      System.exit(0);
    }
  }
```

Exemple de lecture d'un fichier : le parser (5/6)

```
} catch (FileNotFoundException e) {
  e.printStackTrace();
} catch (IOException e) {
  e.printStackTrace();
}

if (nom == null) {
  System.err.println("Le nom n'a pas ete indique");
  System.exit(0);
}
```

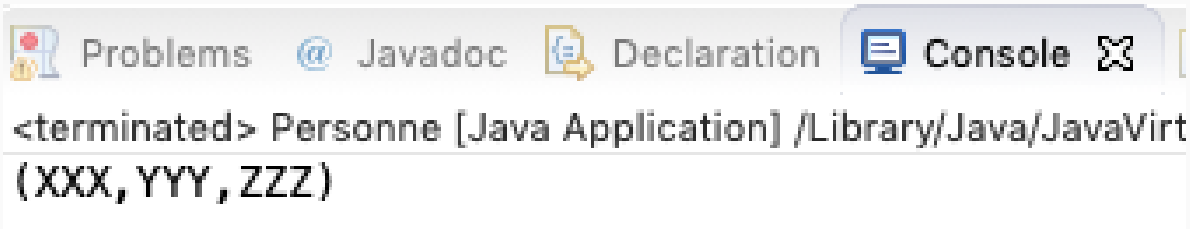
Exemple de lecture d'un fichier : le parser (6/6)

```
        if (prenom == null) {
          System.err.println("Le prenom n'a pas ete"
                                              + " indique");
          System.exit(0);
        }

        if (telephone == null) {
          System.err.println("Le numero de telephone"
                                    + " n'a pas ete indique");
          System.exit(0);
        }
        return new Personne(nom, prenom, telephone);
    }
}
```

Exemple de lecture d'un fichier : le main

```
public static void main(String[] args) {
    Personne p = PersonneParser.parser(
            "/chemin/vers/mon/fichier.txt");
    System.out.println(p);
}
```



Sauvegarder une personne

Pour vous entraîner, utilisez les éléments vus dans le cours pour sauvegarder les données d'une personne dans un fichier

Introduction à la modularité

Modularité : un nouveau niveau d'encapsulation

- Rappel : le principe d'encapsulation consiste à cacher les détails d'implémentation, et à ne donner à l'utilisateur que l'interface dont il a besoin pour utiliser une classe (c'est-à-dire sa partie publique)
- On peut donc modifier la partie privée de la classe (attributs et méthodes) sans impact pour les utilisateurs, tant que la partie publique ne change pas
- Le respect de l'encapsulation garantit la robustesse et la possibilité d'évolution du code

Modularité : l'encapsulation au niveau des packages

- Une « vraie » application va généralement contenir de (très) nombreuses classes
- Les classes doivent normalement être groupées dans des packages « cohérents », c'est-à-dire qui ont un sens logique du point de vue de la conception de l'application
- Exemple : pour un jeu, on peut avoir
  - un package pour les éléments de l'interface graphique
  - un package pour la gestion de la base de donnée (sauvegarde des parties)
  - un package pour l'intelligence artificielle (jeu humain vs machine)
  - ...
- La modularité permet de séparer les packages qui peuvent être utilisés « depuis l'extérieur », et ceux qui ne sont pas accessibles

Module : groupement de packages

- Le principe du module est de regrouper des packages qui ont un lien entre eux : « module ≃ package de packages »
- Dans le module, les packages sont comme d'habitude
- En plus de la structure des packages, le module contient un fichier qui spécifie quelle partie du module est visible depuis l'extérieur
- Les ressources utilisées par un module (images, fichiers de configuration,...) sont liées à ce module, cela parmet une meilleure gestion et distribution du projet, module par module

Créer un module

- À la racine du module, on crée un fichier module−info.java qui décrit le module :

```
module nomDuModule {
  // Configuration du module
}
```

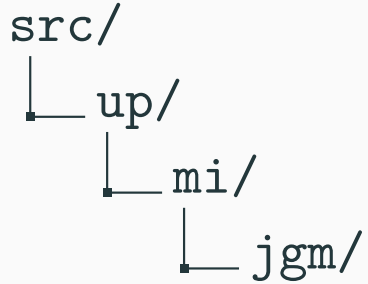
- Informations minimales pour créer un module fonctionnel : son nom
- Convention de nommage :
  - comme pour des packages (minuscules, mots séparés par des points)
  - on recommande que le nom du module corresponde au package « racine » de ce module

```
module up.mi.jgm {
  // Configuration du module
}
```

- On peut ajouter des informations pour décrire finement le comportement du module : on parle de directives

Signification physique d'un module

- Dans le nom d'un package, les points représentent des séparateurs entre dossiers parents et sous-dossiers. Par exemple, le package up.mi.jgm correspond à



Signification physique d'un module

- Dans le nom d'un package, les points représentent des séparateurs entre dossiers parents et sous-dossiers. Par exemple, le package up.mi.jgm correspond à

```
src/
└── up/
    └── mi/
        └── jgm/
```

- Dans le nom d'un module, les points **font partie du nom**. Par exemple, si le package up.mi.jgm est dans le module up.mi.jgm, l'arborescence devient

```
src/
└── up.mi.jgm/
    └── up/
        └── mi/
            └── jgm/
```

Gestion des dépendances : requires

- La directive `requires` permet de spécifier que notre module n'est utilisable que si un autre module est disponible

```
module up.mi.jgm {
    requires un.autre.module ;
}
```

- À la compilation et à l'exécution, le module up.mi.jgm a besoin du module un.autre.module
- Le module up.mi.jgm peut utiliser toute la partie publique de un.autre.module

⚠Pas de dépendances cycliques : si un module A dépend d'un module B, alors B ne peut pas dépendre de A

Gestion des dépendances : requires transitive

- Si up.mi.jgm a besoin de un.autre.module, on peut préciser que les utilisateurs de up.mi.jgm auront également besoin de un.autre.module

```
module up.mi.jgm {
    requires transitive un.autre.module ;
}
```

- Si un autre développeur utilise up.mi.jgm, il n'a pas besoin de préciser que son module a besoin de un.autre.module, c'est fait automatiquement

Rendre publique une partie du module : exports

- Par défaut, les packages dans un module ne sont pas visibles depuis l'extérieur
- Il faut signaler explicitement quels packages seront utilisables depuis d'autres modules

```
module un.autre.module {
    exports un.autre.module.package1;
    exports un.autre.module.package2;
    exports un.autre.module.package3;
}
```

- Si un.autre.module contient un package un.autre.module.package4, ce package ne sera pas utilisable depuis l'extérieur du module

Exporter des sous-packages

- Par défaut, les packages imbriqués dans un package exporté ne sont pas exportés
- Si un.autre.module.package1 est exporté, cela n'exporte pas automatiquement ses sous-packages, il faut les signaler

```
module un.autre.module {
    exports un.autre.module.package1;
    exports un.autre.module.package1.souspackage;
}
```

Exporter des sous-packages

- Par défaut, les packages imbriqués dans un package exporté ne sont pas exportés
- Si un.autre.module.package1 est exporté, cela n'exporte pas automatiquement ses sous-packages, il faut les signaler

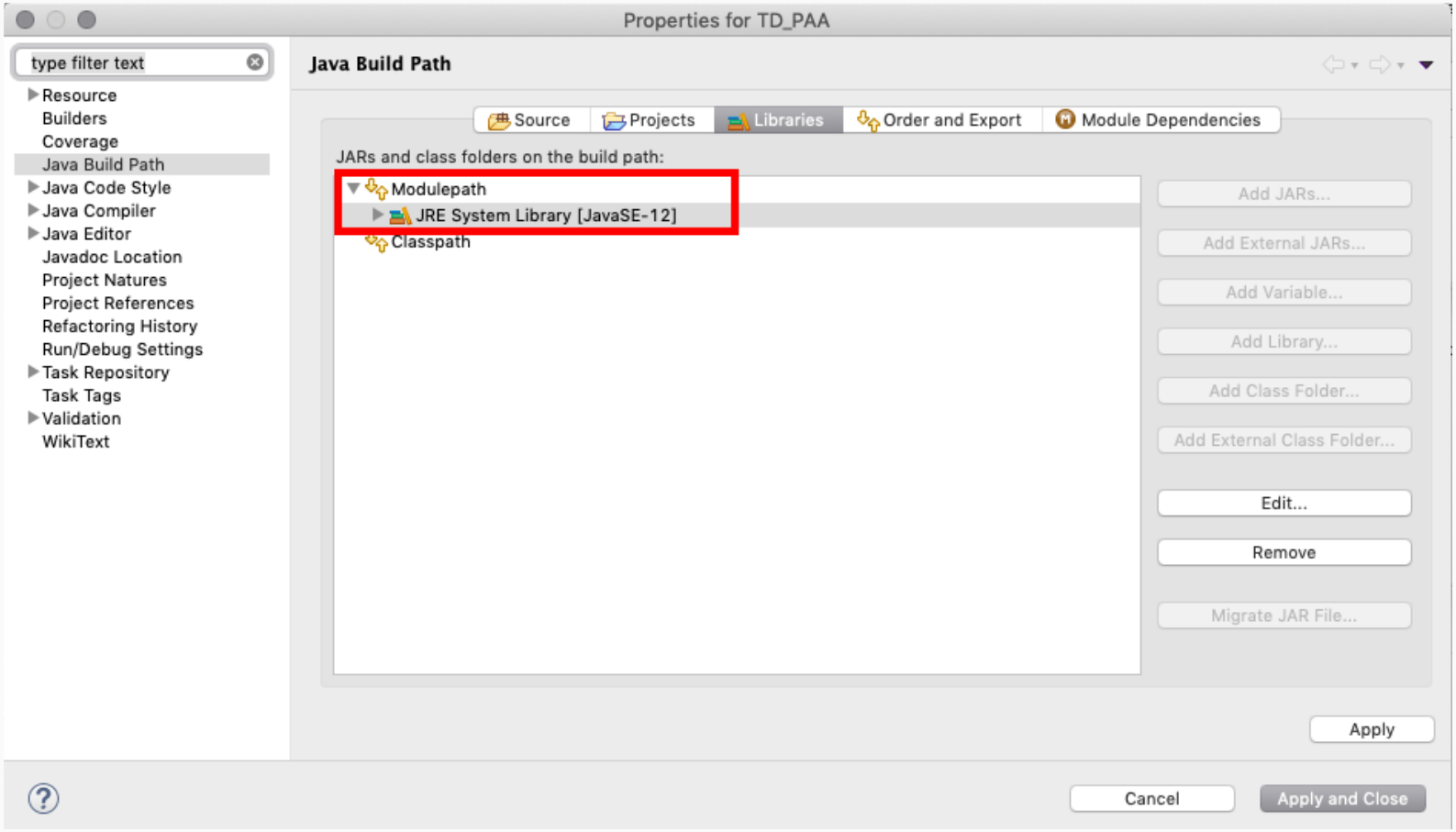
```
module un.autre.module {
    exports un.autre.module.package1;
    exports un.autre.module.package1.souspackage;
}
```

- À l'inverse, il n'est pas nécessaire d'exporter les packages parents pour exporter les sous-packages :

```
module un.autre.module {
    exports un.autre.module.package1.souspackage;
}
```

Utiliser du code Java non modulaire

- On peut utiliser du code Java non modulaire (*e.g.* ≤ Java 8) en plaçant le jar correspondant dans le module path



- Création d'un module automatique qui exporte tous ses packages

Classes internes et lambda expressions

Classe interne en Java

- Une classe interne est une classe définie à l'intérieur d'une classe
- Plusieurs types
  - Classe interne membre
  - Classe locale

Classe interne membre

- Une classe interne membre est définie au même niveau que les autres membres (attributs, constructeurs, méthodes)
- Peut être statique ou liée à une instance
- A un niveau de visibilité (**private**, **protected**, **public**, ou par défaut au niveau du package)

Classe interne membre : Exemple

```
public class UneClasse {
    private int unInt ;
    public String unString ;

    public UneClasse(){
        // ...
    }

    public int uneMethode(){
        // ...
    }

    public class UneClasseMembre{
        // Attributs, methodes, ...
    }
}
```

Nom et import des classes membres

```
package monPackage ;
public class UneClasse {
    // ...

    public class UneClasseMembreInstance{
        // ...
    }

    public static class UneClasseMembreStatique{
        // ...
    }
}
```

- **import** monPackage.UneClasse
- **import** monPackage.UneClasse.UneClasseMembreInstance
- **import** monPackage.UneClasse.UneClasseMembreStatique

Classes membres statiques

- Les classes membres statiques ont accès à tous les membres statiques de la classe englobante

```
package monPackage ;
public class UneClasse {
    private static int unInt = 10 ;

    public static class UneClasseMembreStatique{
        private int total ;
        public UneClasseMembreStatique(int val){
            this.total = val + unInt ;
        }

        public int getTotal(){ return this.total ;}
    }
}
```

Classes membres statiques

- Utilisation dans une autre classe :

```
import monPackage.UneClasse.UneClasseMembreStatique ;

// ...

UneClasseMembreStatique ucms = new
                               UneClasseMembreStatique(5);
System.out.println(ucms.getTotal());
```

- Affiche 15

Classes membres d'instance

- Une instance d'une classe membre d'instance est toujours liée à une instance de la classe englobante

```
public class Banque {
    public Banque(String nom){ ... }
    public String toString(){ ... }

    public class Compte {
        public Compte(String nomTitulaire){ ... }
        public void ajouterOperation(int montant){ ... }
        public String toString(){ ... }
    }
}
```

Classes membres d'instance

```
Banque banque = new Banque("Ma Banque");
Compte compte1 = banque.new Compte("Dark Vador");
Compte compte2 = banque.new Compte("Yoda");
compte1.ajouterOperation(100);
compte2.ajouterOperation(300);
System.out.println(compte1.toString());
System.out.println(banque.toString());
```

- Affiche :

  ```
  Compte :  Dark Vador ; solde : 100 ; Ma Banque
  Banque :  Ma Banque ; capital : 400
  ```

Classes membres d'instance

- La classe membre peut accéder à tous les membres de la classe englobante

```
public class Banque{
    private String nom;
    private long capital = 0;

    public Banque(String nom){this.nom=nom;}

    public String toString(){
        return "Banque : " + nom
          + " ; capital : " + capital ;
    }
```

Classes membres d'instance

```
    public class Compte{
        private String nomTitulaire; private int solde=0;
        public Compte(String nomT){
            nomTitulaire = nomT;
        }
        public void ajouterOperation(int montant){
            this.solde += montant;
            Banque.this.capital += montant;
        }
        public String toString(){
            return "Compte : " + nomTitulaire
                + " ; solde : " + solde
                + " ; " + Banque.this.nom;
        }
    } // Fin de Compte
} // Fin de Banque
```

Classes membres d'instance : syntaxe spécifique

- Création d'une instance de la classe membre d'instance : <instances de la classe englobante>.**new**
  - Exemple : banque.**new** Compte("Dark␣Vador")
- Accès à la classe englobante à partir de la classe interne : <nom de la classe englobante>.**this**
  - Exemple : Banque.**this**

Classes locales

- Classe locale : une classe définie dans une méthode ou un constructeur
- Classe locale anonyme : classe locale définie « à la volée », sans lui donner de nom
- Lambda expression : syntaxe simplifiée pour la définition des classes locales anonymes

Exemple de classe locale : List et Comparator

- Dans l'interface List :
  void sort(Comparator<? **super** E> c)
- E est le type des éléments de la liste
- Les éléments de la liste doivent être comparables via le Comparator spécifié
- L'interface Comparator :

```
public interface Comparator<T>{
/**
 * @param o1 le premier objet a comparer
 * @param o2 le second objet a comparer
 * @return un entier negatif, nul, ou positif
 * si le premier argument est plus petit, egal ou
 * plus grand que le second
 */
int compare(T o1, To2);
}
```

Exemple de classe locale : List et Comparator

- Tri d'une liste en définissant une classe qui implémente Comparator<Integer> :

```
public class Critere implements
                Comparator<Integer>{
  private boolean ordreCroissant;
  public Critere(boolean ordreCroissant){
    this.ordreCroissant = ordreCroissant ;
  }

  @Override
  public int compare(Integer int1, Integer int2){
    return (this.ordreCroissant?1:-1)
                    * int1.compareTo(int2);
  }
}
```

Exemple de classe locale : List et Comparator

- En l'utilisant dans une autre portion de code :

```
List<Integer> list = Arrays.asList(3,17,81,9,12);
list.sort(new Critere(false));
```

- La liste est ainsi triée dans l'ordre décroissant
- Si on n'a besoin du Critere qu'à un seul endroit, il n'est pas nécessaire de définir une « vraie » classe (c'est-à-dire une classe qui n'est pas interne)
- Plusieurs méthodes utilisant les classes locales peuvent s'appliquer

Exemple de classe locale : List et Comparator

- On peut définir une méthode pour trier une liste en fonction d'un critère

```
public static void trier(List<Integer> list,
                         boolean ordreCroissant){
  class Critere implements Comparator<Integer>{
    @Override
    public int compare(Integer int1, Integer int2){
      return (ordreCroissant?1:-1)
                          * int1.compareTo(int2);
    }
  }
  list.sort(new Critere());
}
```

Exemple de classe locale : List et Comparator

```
public static void main(String[] args){
    List<Integer> list = Arrays.asList(3,17,81,9,12);
    trier(list, false);
    System.out.println(list);
    trier(list, true);
    System.out.println(list);
}
```

- Affichage : d'abord [81,17,12,9,3] , ensuite [3,9,12,17,81]
- Avec cette méthode, le Critere n'est jamais utilisé directement, seulement lors de l'appel à trier

Classe locale anonyme

- On peut définir et instancier une classe locale à la volée, sans lui donner de nom

```
public static void trier(List<Integer> list,
                         boolean ordreCroissant){
  list.sort(new Comparator<Integer>{
    @Override
    public int compare(Integer int1, Integer int2){
      return (ordreCroissant?1:-1)
                          * int1.compareTo(int2);
    }
  }
  );
}
```

Le point sur les classes locales

- Les classes locales (anonymes ou nommées) ont des points communs avec les classes membres
  - Une instance de la classe local est toujours liée à la classe englobante
  - La classe locale a accès à tous les membres de la classe englobante
- La classe locale a accès à toutes les variables **final** de sa zone de portée
- La classe locale est accessible dans le bloc où elle est définie, juste après sa définition

Lambda expression

- Lambda expression : simplification syntaxique pour les classes locales anonymes qui n'implémentent qu'une seule méthode

```
public static void trier(List<Integer> list,
                         boolean ordreCroissant){
  list.sort( (int1, int2) -> {
    return (ordreCroissant?1:-1)
                          * int1.compareTo(int2);
  }
  );
}
```

- De manière générale, la syntaxte est (parametres) -> {bloc d'instructions} avec la liste des paramètres et le corps de la méthode de la classe locale