Cours 6

I. Abstraction

I.1 Classe abstraite

I.1.a `DBShape s = new DBShape( ); s.draw( );` --> drawSpec( ) "bidon"

I.1.b interdire l'instanciation, mais laisser en commun ce qui doit l'être; classe prévue pour être dérivée

I.1.c constructeurs : utiles pour initialiser les attributs communs; appelés par `super(...);`

I.1.d syntaxe: `abstract` (public abstract class NomClasse ...)

I.2 Méthode abstraite

I.2.a interdire l'exécution ==> pas de corps, évite les corps "bidon" (drawSpec( ))

I.2.b redéfinition obligatoire dans toutes les sous-classes (sinon elles seront abstraites) ==> ne concerne pas les constructeurs, private interdit, `@Override` .
Garantit une cohérence, un contrat avec les sous-classes, y compris les futures ==> extensibilité.

I.2.c syntaxe: `abstract` et `;` au lieu de `{}` (public abstract void nomMethode( ) ;)

I.2.d Si dans une classe il existe une méthode abstraite (non redéfinie), cette classe est abstraite (==> `abstract`) car `objet.methode()` doit toujours pouvoir marcher si objet a pu être instancié

I.2.e draw() peut quand-même appeler drawSpec() grâce à la liaison dynamique ==> utilisation de classes non encore écrites

I.2.f Il est évident qu'une classe non abstraite peut hériter d'une classe abstraite, mais il est aussi possible qu'une classe abstraite hérite d'une classe non abstraite.

I.3 Interface (pour compenser l'absence d'héritage multiple)

I.3.a et si 100% des méthodes étaient abstraites ? et aucune variable d'instance ?

I.3.b 100% des méthodes `public abstract` ==> 2 mots inutiles, et pas de constructeurs

I.3.c 100% des attributs `public static final` ==> 3 mots inutiles

I.3.d 100% des méthodes à ~~re~~définir ==> pas de `@Override`

I.3.e syntaxe: `interface` au lieu de `abstract class` , `implements` au lieu de `extends` , une classe peut implémenter plusieurs interfaces (séparées par des virgules) ==> implémenter toutes les méthodes de toutes les interfaces !

I.3.f exemple d'interface: `Comparable` ==> `compareTo()` -- `Circle extends DBShape` -- `DBShape implements Comparable, Drawable` -- `Rosace implements Drawable`

I.3.g exemple d'objet: `Drawable obj = new Circle(); obj.draw();` plutôt que `Circle obj = new Circle();`

I.3.h héritage d'interfaces: si `interface SousI extends SuperI` , on devra implémenter toutes les méthodes de `SousI` et `SuperI`
héritage multiple !

I.3.i véritable type ==> l'opérateur `instanceof` fonctionne; comme pour une super-classe (abstraite ou non) : *`réfObjet`* `instanceof` *`ClasseOuInterface`*

II. Exceptions

II.1 Résumé de cours

II.2 **Essayer** la classe de démonstration et la comprendre

II.3 Hiérarchie des exceptions du paquetage `java.lang`

II.4 Tutoriel (en anglais)

Lire le poly :

tout jusqu'à la section 2.4.1, **2.4.2**, sections 2.5, 3.1, **3.2**, 3.2.2, **3.3**, 3.6, 4, 5.1, 5.2.0, 6, 7.1, 7.2, 8.1, 8.2.1.1, 8.2.2, 9.1 à 9.5, **9.6, 9.7, 10**, 13.3, et annexes 6 & 7

Exceptions

1) Exemple sans exception

```
public double get( int i ) { return tab[i-1]; }
```
que retourner si mauvais indice ?

2) Définition

2.1. Une exception est un évènement exceptionnel qui peut survenir pendant le déroulement d'un programme et qui provoque le déroutement du flot d'instructions normal.
2.2. En java, une exception est un objet d'une sous-classe d'Exception et des exceptions de types différents peuvent être distinguées.
2.3. On peut créer ses propres exceptions et y ajouter de l'information.

3) Système de gestion des exceptions

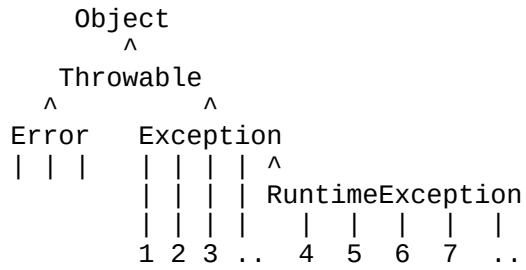
3.1. "parallèle" au déroulement normal; lancer une exception et la rattraper ailleurs
3.2. Exemple
main ==> proc A ==> proc B ==> func C
func C lance une exception --> proc B
proc B n'a rien prévu pour la traiter --> proc A
proc A sait traiter l'exception et continue son exécution
main ne s'est aperçu de rien
si rien n'est prévu nulle part, le programme se termine avec un long message (StackTrace).

4) Avantages

4.1. séparer les instructions normales du traitement des erreurs
4.2. propager l'exception à la méthode appelante et ainsi de suite
4.3. différencier et regrouper des exceptions différentes grâce aux hiérarchies de classes (héritant de Exception)

5) Exceptions existant dans Java

5.1. Hiérarchie



5.2. Error est un type d'erreur système "grave"; on ne peut rien faire, donc inutile d'essayer de les attraper et traiter.
(exemples: AssertionEr, VirtualMachineEr, OutOfMemoryEr, StackOverflowEr)
5.3. Exception est un type d'erreur qui peut être traitée; on a le choix de dire qu'une méthode la lance ou bien de l'attraper et traiter à l'intérieur.
(catch or specify, checked exceptions, outside JVM)
(exemples: CloneNotSupportedEx, AWTEx, IOEx, EOFEx, FileNotFoundEx)
5.4. RuntimeException est un type particulier d'exception qui correspond en général à une erreur du programmeur; on n'a donc pas obligation ni de la déclarer ni de l'attraper et traiter. Par contre, on a obligation de corriger son programme pour qu'elle n'arrive pas !
(unchecked exceptions, inside JVM)
(exemples: ArithmeticEx, ClassCastEx, NegativeArraySizeEx, NullPointerEx, IllegalArgumentEx,

NumberFormatEx, IndexOutOfBoundsEx)

5.5. Les classes 1 2 ... correspondent aux exceptions déjà définies dans les nombreuses classes Java.

6) Traitement des exceptions lancées par le JDK

6.1. Exemple:
`try { instructions } catch ( UneClasseException pObjetException ) { traitement }`
(voir méthodes `getStackTrace(), printStackTrace(), getMessage(), getLocalizedMessage(), toString()`)

6.2. S'il y a plusieurs exceptions possibles, on peut mettre successivement plusieurs catch après un seul try.
Attention! l'ordre est important (du particulier au général)

6.3. Si un traitement est commun à plusieurs exceptions héritant d'une même classe, écrire `catch ( ClasseMereException objetException )`

6.4. On sait quelle classe exception en regardant la javadoc
Exemple: **Throws:** `UneClasseException`

6.5. Si une méthode appelée lance une exception dont la classe n'hérite pas de `RunTimeException`, soit il faut entourer l'appel d'un `try/catch`, soit il faut déclarer qu'on lance l'exception
==> `throws UneClasseException` à la fin de la signature (plusieurs possibles)

6.6. On peut traiter partiellement et relancer l'exception dans le `catch`: `throw e;`
==> `throws UneClasseException`

6.7. Si un traitement doit être fait qu'une exception soit lancée ou pas, ajouter après le `try/catch` un bloc `finally { traitement }`
(si exception imprévue, ou return dans try ou catch)
Classe de démonstration du bloc `finally` (à essayer !)

7) Lancer des exceptions du JDK

7.1. Exemple: `if (mauvais paramètre) throw new UneClasseException("...");`
==> throws UneClasseException

7.2. Ne pas rattraper une exception dans la même méthode que celle où on l'a lancée; il vaut mieux
`if (cond) { i1 } else { i2 }` que
`try { if (!cond) throw new UneClasseException(); i1 } catch ( UneClasseException e ) { i2 }`

8) Créer ses propres exceptions

8.1. Créer une classe qui extends Exception ou une de ses sous-classes

8.2. Redéfinir éventuellement getMessage()

8.3. Ajouter éventuellement des attributs (donc constructeur), voire des méthodes

9) Les assertions

9.1. Programmation par contrat, sécurité : pré-condition, post-condition, pas d'invariant

9.2. `assert condition : message;`
équivalent à `if (!condition) throw new AssertionError(message);`
mais comment les retirer ?

9.3. Pour autoriser les assertions :
`-ea` à la compilation (difficile en BlueJ, voir `javac`)

Hierarchy For Package java.lang

- java.lang.**Object**

Class Hierarchy for (unchecked) exceptions

- java.lang.**Throwable** (implements java.io.Serializable)
  - java.lang.**Error**
    - java.lang.**AssertionError**
    - java.lang.**LinkageError**
      - java.lang.**ClassCircularityError**
      - java.lang.**ClassFormatError**
        - java.lang.**UnsupportedClassVersionError**
      - java.lang.**ExceptionInInitializerError**
      - java.lang.**IncompatibleClassChangeError**
        - java.lang.**AbstractMethodError**
        - java.lang.**IllegalAccessError**
        - java.lang.**InstantiationError**
        - java.lang.**NoSuchFieldError**
        - java.lang.**NoSuchMethodError**
      - java.lang.**NoClassDefFoundError**
      - java.lang.**UnsatisfiedLinkError**
      - java.lang.**VerifyError**
    - java.lang.**ThreadDeath**
    - java.lang.**VirtualMachineError**
      - java.lang.**InternalError**
      - java.lang.**OutOfMemoryError**
      - java.lang.**StackOverflowError**
      - java.lang.**UnknownError**
  - java.lang.**Exception** <----- voir "Direct Known Subclasses", en particulier IOException
    - java.lang.**ClassNotFoundException**
    - java.lang.**CloneNotSupportedException**
    - java.lang.**IllegalAccessException**
    - java.lang.**InstantiationException**
    - java.lang.**InterruptedException**
    - java.lang.**NoSuchFieldException**
    - java.lang.**NoSuchMethodException**
    - java.lang.**RuntimeException**
      - java.lang.**ArithmeticException**
      - java.lang.**ArrayStoreException**
      - java.lang.**ClassCastException**
      - java.lang.**EnumConstantNotPresentException**
      - java.lang.**IllegalArgumentException**
        - java.lang.**IllegalThreadStateException**
        - java.lang.**NumberFormatException**
      - java.lang.**IllegalMonitorStateException**
      - java.lang.**IllegalStateException**
      - java.lang.**IndexOutOfBoundsException**
        - java.lang.**ArrayIndexOutOfBoundsException**
        - java.lang.**StringIndexOutOfBoundsException**
      - java.lang.**NegativeArraySizeException**
      - java.lang.**NullPointerException**
      - java.lang.**SecurityException**
      - java.lang.**TypeNotPresentException**
      - java.lang.**UnsupportedOperationException**

For further API reference and developer documentation, see Java SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.