Hiérarchie de classes et exceptions

Héritage

Attention ¶ §

L'héritage possède deux caractéristiques pouvant parfois être en contradiction:

- l'**extension de code**, qui consiste à dériver une nouvelle classe d'une classe déjà existante, pour hériter du code de la classe existante et en ajouter sans avoir à tout réécrire.
- le **polymorphisme** qui modélise la relation dans laquelle un objet peut être utilisé à la place d'un autre objet.

Syntaxe et vocabulaire

```
class B extends A { ... }
```

```
class C extends A { ... }
```

A	B, C
base	dérivée
mère, parente	fille
super-classe	sous-classe
qui est hérité	qui hérite
qui est étendue	qui étend
type, sur-type	sous-type

Extension de code

Principe

La classe dérivée hérite des membres non marqués **private** de la classe de base (mis à part les constructeurs). Cette classe peut elle-même posséder des champs et méthodes supplémentaires. C'est le principe de l'extension, suggéré par le mot-clé `extends`.

Le mot-clé **protected** au-devant des membres de la classe de base permet d'élargir leur accès à toutes les classes dérivées.

Ex.1. Point/Vecteur (5 min)

Observez les fichiers **`Point.java`** et **`Vecteur.java`**.

- Que fait ce code ?
- A quoi correspondent `x` et `y` dans chacun des fichiers ?
- A quoi sert le mot-clé `super` ?
- Voyez-vous un cas de **surcharge** (= *overloading*), qui permet à plusieurs méthodes de partager le même *nom*, à condition que leur *signature* (nombre et type des paramètres) soit différente ?

Ex.2. Complexe (10 min)

Faites une classe `Complexe`, qui étend `Vecteur`, par les méthodes:

- `double obtenirNorme()`
- `Complexe obtenirConjugue()`
- `void multiplier(Complexe c)`

NB: pour $z = x + iy$, $z' = x' + iy'$,

- $Nz = x^2 + y^2$ (norme),
- $\bar{z} = x - iy$ (conjugué),
- $z * z' = (xx' - yy') + i(xy' + yx')$.

Ce qu'il faut retenir

- Les classes sont organisées en une hiérarchie. Le mot-clé `extends` indique qu'une classe descent d'une autre.
- L'état et le comportement associés aux instances d'une classe sont automatiquement partagés à toute classe d'un descendant (propriété d'extension de code).
- Dans une classe, le mot-clé `this` permet d'adresser des requêtes à soi-même, tandis que le mot-clé `super` permet d'adresser des requêtes à son parent.

Polymorphisme

Principe

L'héritage modélise aussi la relation dans laquelle un objet de la classe dérivée peut être utilisé comme un objet de la classe de base, c'est ce qui fait de l'héritage un mécanisme complexe.

```
class B extends A { ... }
```

```
B objetB = new B();
A objetA = new B(); //transtypage ascendant implicite
```

Un objet de la classe B *est un* objet de la classe A et peut être utilisé comme tel.

Attention: cette relation n'est pas *symétrique*.

Requêtes

- Une méthode `methodeA` non privée de la classe `A`, peut être appelée à partir de la variable `objetA`:

```
objetA.methodeA(); //compile
objetB.methodeA(); //compile (extension de code)
```

- Si une méthode `methodeB` n'est définie que dans la classe `B`, on ne peut l'appeler à partir de la variable `objetA`:

```
objetA.methodeB(); //ne compile pas (objetA est de type A)
objetB.methodeB(); //compile (objetB est de type B)
```

Liaison dynamique

A l'exécution, la machine virtuelle choisit la méthode à appeler en réponse à une requête, c'est le principe de la **liaison dynamique**.

La recherche de cette méthode commence avec la classe de l'objet auquel la requête est adressée. Si aucune méthode appropriée n'est trouvée, la recherche se poursuit dans la classe parente et ainsi de suite jusqu'à ce qu'une méthode soit trouvée (le compilateur a préalablement vérifié qu'il y aura toujours ultimement une méthode appropriée).

Transtypage

Vous connaissez le transtypage ascendant (= *upcast*) implicite:

```
A objetA = new B(); //transtypage ascendant implicite
```

A l'inverse, il est possible de réaliser explicitement un transtypage descendant (= *downcast*):

```
B objetB2 = (B) objetA; //transtypage descendant explicite
```

C'est utile quand on manipule une instance de `B` comme un `A` (passage de paramètres par exemple), mais qu'on a besoin d'appeler `methodeB`.

Ex.3. TestComplexe (5 min)

Ecrivez une classe `TestComplexe`, dans laquelle vous testez

- la cohérence de l'addition et de la soustraction des nombres complexes en appelant directement la méthode `testsUnitaires` de la classe **`TestVecteur.java`**.
- la cohérence de la multiplication avec la norme et la conjugaison (la partie réelle de $z\bar{z}$ doit être égale à la norme Nz).

Redéfinition

Et si une même méthode `methodeAB` est définie à la fois dans `A` et `B` ?

Dans une classe fille, il est possible de redéfinir certaines méthodes dont elle hérite pour les implémenter d'une autre manière. En réponse à un appel à `methodeAB` adressé à `objetB`, ce sera la code de la classe `B` qui sera exécuté (et non celui de la classe `A`).

Ne pas confondre **redéfinition** (= *overriding*), même signature, mais corps différent entre la classe de base et la classe dérivée, et **surcharge** (= *overloading*), même nom, mais liste de paramètres différente, au sein d'une même classe.

Hiérarchie de classes

Rien n'empêche de dériver une classe, elle-même dérivée d'une autre classe et ainsi de suite. L'héritage est *transitif*: si `B` hérite de `A` et si `D` hérite de `B`, alors `D` hérite aussi de `A` via `B`.

En Java, toutes les classes dérivent par défaut de `java.lang.Object` (cf. l'API standard). Cette classe possède quelques méthodes pouvant être redéfinies comme `toString` qui retourne une représentation textuelle de type `String` de l'objet (nom de la classe, arobase, hash code par défaut).

Ex.4. Notation complexe (5 min)

Redéfinissez la méthode `toString` dans votre classe `Complexe` de façon à afficher les nombres en notation complexe (sous la forme $x + iy$), plutôt qu'en notation vectorielle (sous la forme (x, y)).

Ce qu'il faut retenir

- ce que c'est qu'une **surcharge** (dans une classe, plusieurs méthodes ayant le même nom, mais une signature différente) et une **redéfinition** (une classe et ses descendantes ont chacune une méthode identique),
- ce que c'est que le **polymorphisme** (toutes les instances d'une classe peuvent être vus comme des instances d'une classe parente),
- le mécanisme de **liaison dynamique** (comment la machine virtuelle recherche à l'exécution la méthode à appeler en réponse à une requête).

Exceptions

Erreurs et exceptions

Les **exceptions** désignent les situations où l'exécution peut se poursuivre, généralement de façon différente. Elles sont matérialisées en Java par des instances de classes dérivant de `java.lang.Exception`, elle-même dérivant de `java.lang.Throwable`.

C'est donc aussi l'occasion d'avoir un aperçu de la hiérarchie des classes de l'API standard:

java.lang.Object
    java.lang.Throwable
        java.lang.Exception

N'hésitez pas à lire les tutoriaux qui traitent le sujet.

Le développeur

Le développeur d'une classe peut indiquer aux clients qu'une méthode est susceptible de lever une exception avec le mot-clé `throws` et peut effectivement **lever une exception** au moment voulu avec le mot-clé `throw`.

```
     public int pop() throws Exception {
         if ( myNode == null )
             throw new Exception();
         else
             myNode = myNode.next();
     }
```

Créer sa propre classe d'exception

```
public class EmptyStackException extends Exception {
    ...
}
```

```
public int pop() throws EmptyStackException {
    if ( myNode == null )
        throw new EmptyStackException();
    else
        myNode = myNode.next();
}
```

Propager une exception

```
private static void oneMove(Stack src, Stack dest)
```

```
  throws EmptyStackException {
    try {
        dest.push( src.top() );
        src.pop();
    } catch (EmptyStackException e) {
        throw new EmptyStackException("empty stack");
    }
}
```

Plutôt que d'attraper et lever la même exception, il est possible de la **propager**.

```
private static void oneMove(Stack src, Stack dest)
  throws EmptyStackException {
    dest.push( src.top() );
    src.pop();
}
```

Le client qui traite les exceptions

Le bloc d'instructions principal est mis dans un bloc `try`, tandis que la gestion des exceptions est répartie, selon la nature de l'exception, dans des blocs `catch` successifs.

```
try {
    /* code */
} catch(ExceptionDeTypeA e) {
    /* gestion des exceptions de type A */
} catch(ExceptionDeTypeB e) {
    /* gestion des exceptions de type B */
} finally {
    /* tout fermer et nettoyer */
}
```

Le bloc optionnel `finally` s'exécute toujours.

Celui qui n'en fait pas assez

Ne jamais écrire un code qui masque les exceptions.

```
//PAS BIEN
try {
  unCodeQuiLeveUneException();
} catch(Exception e) {
  /* Aucune action, ce qui masque les erreurs */
}
```

Préférez au moins:

```
try {
  unCodeQuiLeveUneException();
} catch(Exception e) {
  /* affiche l'empilement des appels qui ont mené à l'erreur */
```

```
  e.printStackTrace();
}
```

Celui qui en fait trop

N'entourez pas chaque instruction d'un bloc `try`/`catch`: ça ne sert à rien et va à l'encontre de l'objectif qui est de **séparer** le bloc d'instructions principal, des instructions relevant de la gestion des exceptions pouvant survenir dans ce bloc, afin d'obtenir un code plus lisible et plus facile à réutiliser.

```
//PAS BIEN
try {
  unCodeQuiLeveUneExceptionA();
} catch(ExceptionA e) {
  e.printStackTrace();
}
try {
  unCodeQuiLeveUneExceptionB();
} catch(ExceptionB e) {
  e.printStackTrace();
}
```

Ex.5. Exceptions (10 min)

Dans votre classe `Complexe`, ajoutez la méthode suivante:

`void diviser(Complexe c)` (NB. $z/z' = z\bar{z'}/Nz'$)

Dans une nouvelle classe `DemoComplexe`, appelez cette méthode avec en paramètre un complexe nul ($0 + i0$). En l'affichant sur la sortie standard, vérifiez que le résultat n'est pas défini.

Dans votre classe `Complexe`, levez vous-même une exception personnalisée `DivisionComplexeParZero` et attrapez-là dans `DemoComplexe`.

Ce qu'il faut retenir

- Les exceptions sont des instances de classes dérivant de `java.lang.Exception`.
- La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc `catch` acceptant cette exception soit trouvé.
- L'appel à une méthode susceptible de lever une exception doit :
  - soit être contenu dans un bloc `try` / `catch`
  - soit être situé dans une méthode propageant cette classe d'exception (`throws`)
- Un bloc `finally` peut suivre les blocs `catch`. Son contenu est toujours exécuté (avec ou sans exception, et même en cas de `break`, `continue`, `return` dans le bloc `try`).