Cours Java_II

Cnam Paris
jean-michel Douin, douin@cnam.fr
version :6 sept 2004

Notes de cours java : le langage : Une classe, constructeurs, surcharge

Les notes et les Travaux Pratiques sont disponibles en
http://jfod.cnam.fr/tp_cdi/douin/ et http://jfod.cnam.fr/tp_cdi/
Ce support ne représente que des notes de cours, il ne peut se substituer à la lecture d'un ouvrage sur ce thème; Le livre de Liskov cité en bibliographie en est un exemple.

Sommaire

- Classe syntaxe
- Création d'instances
  - Ordre d'initialisation des champs
- Constructeur
- Surcharge
- Encapsulation
  - Règles de visibilité
  - Paquetage
- Classes imbriquées
- Interfaces
- Classes incomplètes/abstraites
- Classe et exception

Bibliographie utilisée

- The Java Handbook, Patrick Naughton. Osborne McGraw-Hill.1996.
  http://www.osborne.com

- Thinking in Java, Bruce Eckel, http://www.EckelObjects.com

- Data Structures and Problem Solving Using Java. Mark Allen Weiss. Addison Wesley http://www.cs.fiu.edu/~weiss/#dsj

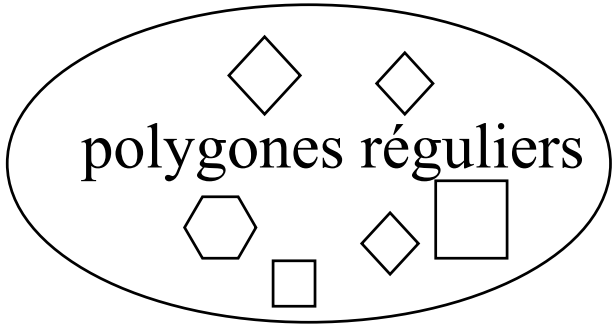
- http://java.sun.com/docs/books/jls/

- http://java.sun.com/docs/books/tutorial.html

- Program Development in Java,
  Abstraction, Specification, and Object-Oriented Design, B.Liskov avec J. Guttag
  voir http://www.awl.com/cseng/ Addison Wesley 2000. ISBN 0-201-65768-6

Concepts de l'orienté objet

- **Classe et objet (instance)**
- **Etat d'un objet et encapsulation**
- **Comportement d'un objet et méthodes**



Classe : Syntaxe

```
class  NomDeClasse {
    type  variable1DInstance;
    type  variable2DInstance;
    type  variableNDInstance;

    type nom1DeMethode ( listeDeParametres) {

    }
    type nom2DeMethode( listeDeParametres) {

    }
    type nomNDeMethode( listeDeParametres) {

    }
}
```

Classe : Syntaxe et visibilité

- **visibilité** class NomDeClasse {

  visibilité type variable1DInstance;
  visibilité type variable2DInstance;
  visibilité type variableNDInstance;

  visibilité type nom1DeMethode (listeDeParametres) {

  }
  visibilité type nom2DeMethode(listeDeParametres) {

  }
  visibilité type nomNDeMethode(listeDeParametres) {
  }
- }

visibilité ::= public | private | protected | < vide >

Exemple: la classe des polygones réguliers

```
public class PolygoneRegulier{
  private int nombreDeCotes;
  private int longueurDuCote;

  void initialiser( int nCotes, int longueur){
    nombreDeCotes = nCotes;
    longueurDuCote = longueur;
  }

  int perimetre(){
    return nombreDeCotes * longueurDuCote;
  }

  int surface(){
    return (int) (1.0/4 * (nombreDeCotes *
Math.pow(longueurDuCote,2.0) *
                  cotg(Math.PI / nombreDeCotes)));
  }

  private static double cotg(double x){return Math.cos(x) / Math.sin(x); }
}
```

données d'instance

méthodes d'instance

méthode de classe

Création et accès aux instances

- **Declaration d'instances et création**

  PolygoneRegulier p; *// ici p == null*

  p = new PolygoneRegulier (); *// p est une référence sur un objet*

  PolygoneRegulier q = new PolygoneRegulier (); *// en une ligne*

- **La référence d'un objet est l' adresse**

  d'une structure de données contenant des informations sur la classe declarée

  à laquelle se trouvent les champs d'instance et d'autres informations

  (cette référence ne peut être manipulée)

- **Opérateur "."**

  appels de méthodes (si accessibles)

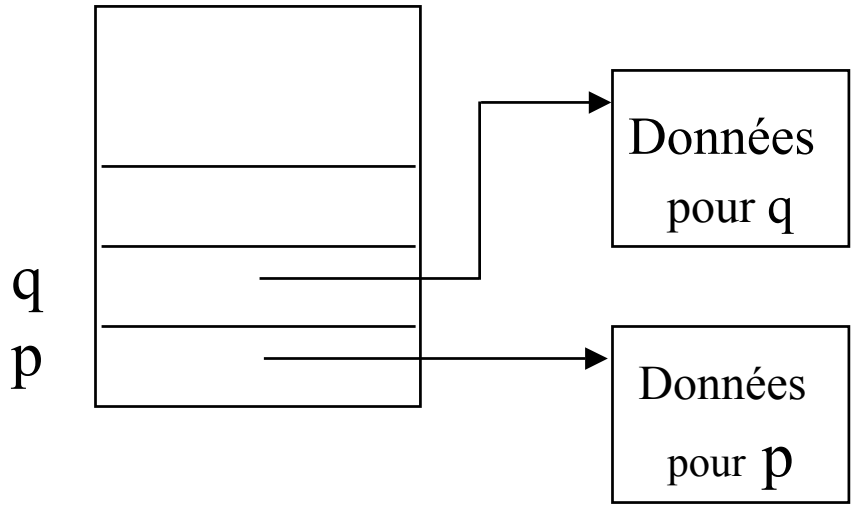
  accès aux champs (si accessibles)

- **en passage de paramètre**

  par valeur, soit uniquement la référence de l'objet

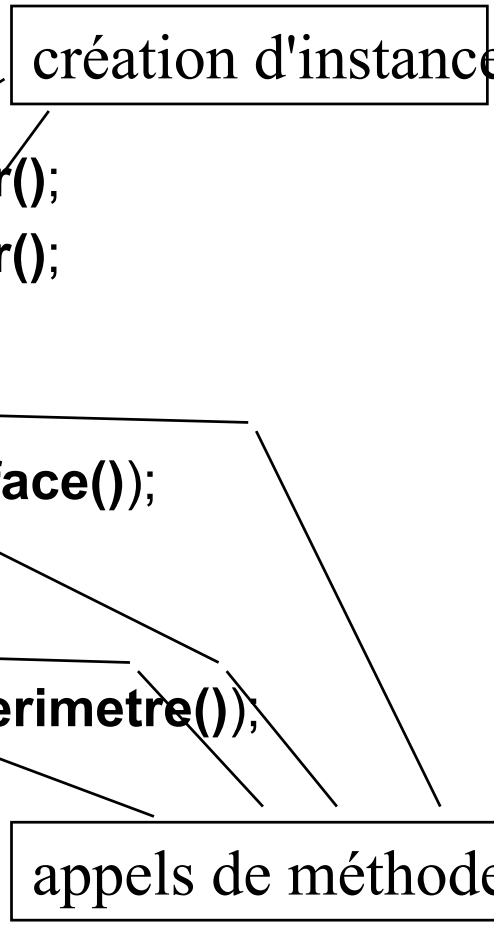
Instances et mémoire

p = new PolygoneRegulier (); *// p est une référence sur un objet*
PolygoneRegulier q = new PolygoneRegulier (); *// en une ligne*



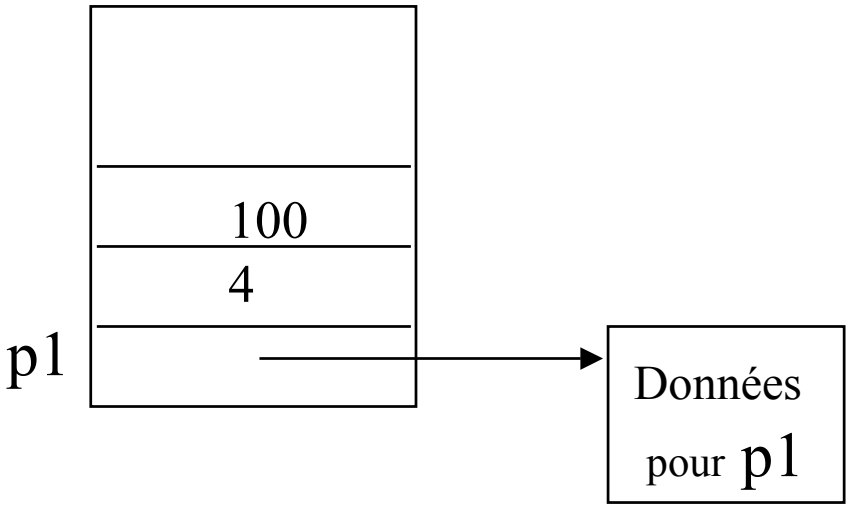
Utilisation de la classe

```
public class TestPolyReg{

  public static void main(String args[]){
    PolygoneRegulier p1 = new PolygoneRegulier();
    PolygoneRegulier p2 = new PolygoneRegulier();

    p1.initialiser(4,100);
    System.out.println(" surface de p1 = " + p1.surface());

    p2.initialiser(5,10);
    System.out.println(" perimetre de p2 = " + p2.perimetre());

  }
```



Instances et appels de méthodes

p1.initialiser(4,100);



Pile d ’éxecution
avant l ’appel de la
méthode initialiser

Initialisation des variables d'instance

- **champs d'instance initialisés**
  - **à 0 si de type entier, 0.0 si flottant, 0 si char, false si booléen**
  - **à null si référence d'objet**

- **Variables locales aux méthodes**
  - **elles ne sont pas initialisées, erreur à la compilation si utilisation avant affectation**

Constructeur

- public class **PolygoneRegulier{**
- private int nombreDeCotes;
- private int longueurDuCote;

- **PolygoneRegulier (int nCotes, int longueur) {** **// void initialiser**
- nombreDeCotes = nCotes;
- longueurDuCote = longueur;
- }

- **....**

- **le constructeur a le même nom que la classe**

PolygoneRegulier P = new PolygoneRegulier(4, 100);

A la création d'instance(s) les 2 paramètres sont maintenant imposés : le seul constructeur présent impose son appel

Constructeur par défaut

- public class **PolygoneRegulier**{
- private int nombreDeCotes;
- private int longueurDuCote;

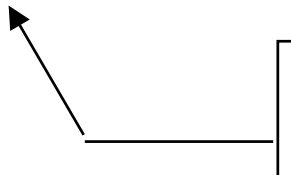
- void initialiser(int nCotes, int longueur){...}

- int perimetre(){...}

- int surface(){....}

- }

- public static void main(String args[]){
- PolygoneRegulier p1 = new PolygoneRegulier();
- PolygoneRegulier p2 = new PolygoneRegulier();



Destructeurs et ramasse miettes

- **pas de destructeurs (accessibles à l'utilisateur)**
- **La dé-allocation n'est pas à la charge du programmeur**
- **Le déclenchement de la récupération mémoire dépend de la stratégie du ramasse miettes**

(voir java.lang.System.gc())

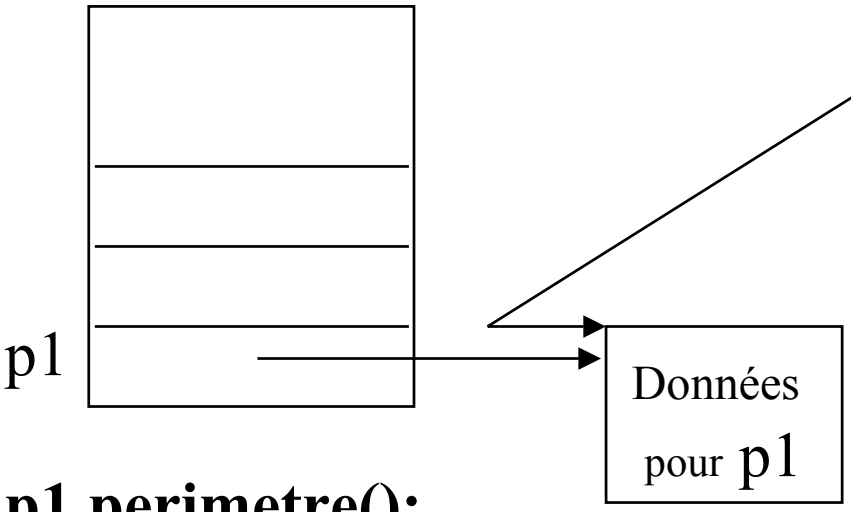
Garbage collection, Richard Jones and Rafael Lins. Wiley
http://www.ukc.ac.uk/computer_science/Html/Jones/gc.html

La référence this

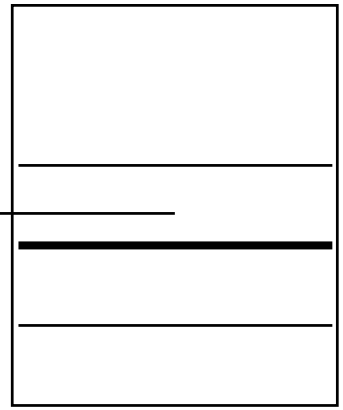
```
public class PolygoneRegulier{
  private int nombreDeCotes;
  private int longueurDuCote;

public PolygoneRegulier ( int nombreDeCotes, int longueur){
      this.nombreDeCotes = nombreDeCotes;
      this.longueurDuCote = longueur;
  }

  public int perimetre(){
    return this.nombreDeCotes *  this.longueurDuCote;
  }
```

this et l'appel d 'une méthode

```
public int perimetre(){
  return this.nombreDeCotes *  this.longueurDuCote;
 }
```



p1.perimetre();
Pile d 'éxecution
avant l 'appel de la
méthode perimetre



p1.perimetre();
Pile d 'éxecution
pendant l 'appel de l
méthode perimetre

Surcharge, polymorphisme ad'hoc

- **Polymorphisme ad'hoc**

  Surcharge(overloading),

  plusieurs implémentations d'une méthode en fonction des types de paramètres souhaités, le choix de la méthode est résolu statiquement dès la compilation

  Opérateur polymorphe : 3 + 2; 3.0 + 2.0, "bon" + "jour"

```
public class java.io.PrintWriter ..... {
    ....
    public void print(boolean b){...};
    public void print(char c){...};
    public void print(int i){...};
    public void print(long l){...};
    ....
}
```

Surcharge, présence de plusieurs constructeurs

- public class PolygoneRegulier{
- private int nombreDeCotes;
- private int longueurDuCote;

- **PolygoneRegulier (int nCotes, int longueur) {**
- nombreDeCotes = nCotes;
- longueurDuCote = longueur;
- }

- **PolygoneRegulier () {**
- nombreDeCotes = 0;
- longueurDuCote = 0;
- }
- .....

- public static void main(String args[]){
- PolygoneRegulier p1 = new **PolygoneRegulier(4,100)**;
- PolygoneRegulier p2 = new **PolygoneRegulier()**;

Surcharge (2)

- public class PolygoneRegulier{
- private int nombreDeCotes;
- private double longueurDuCote;

- **PolygoneRegulier (int nCotes, int longueur) {**
- nombreDeCotes = nCotes;
- longueurDuCote = longueur;
- }

- **PolygoneRegulier (int nCotes, double longueur) {**
- nombreDeCotes = nCotes;
- longueurDuCote = longueur;
- }

- **PolygoneRegulier () {....** }

- public static void main(String args[]){
- PolygoneRegulier p1 = new **PolygoneRegulier(4,100)**;
- PolygoneRegulier p2 = new **PolygoneRegulier(5,101.6)**;

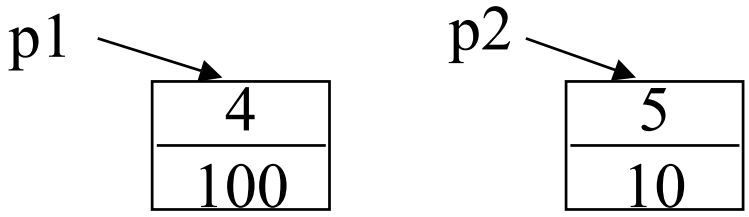
Un autre usage de this

```
public class PolygoneRegulier{
  private int nombreDeCotes;
  private int longueurDuCote;

   PolygoneRegulier  ( int nCotes, int longueur) {
      nombreDeCotes = nCotes;
      longueurDuCote = longueur;
    }

   PolygoneRegulier  () {
      this( 1, 1);                  // appel du constructeur d'arité 2
    }

}
```

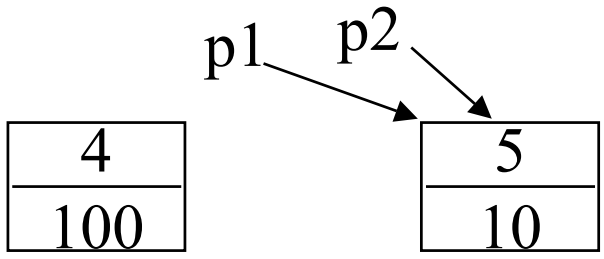
Affectation

- affectation de références =

- public static void main(String args[]){
- PolygoneRegulier p1 = new **PolygoneRegulier(4,100)**;
- PolygoneRegulier p2 = new **PolygoneRegulier(5,10)**;



- p1 = p2



Affectation : attention

- **Affectation entre deux variables de types primitifs :**

  Affectation des valeurs : int i = 3; int j = 5; i=j;

- **Affectation entre deux instances de classes :**

  Synonymie : Polygone p1, p2; … p1=p2;

  p1 et p2 sont deux noms pour le même objet

- **Contenu et contenant**

  A l ’occasion d ’une promenade dominicale au bord d ’une rivière !

  Peut-on dire que cette rivière est toujours la même alors que ce n ’est jamais la même eau qui y coule ?

  (en java, pour une instance, c ’est toujours la même rivière)

Méthodes : une classification(1)

- **Constructeur**

- **Accesseur**

  Méthodes de lecture

  (famille *get*xxxxx*)*

- **Mutateur**

  Méthodes modifiant l'état d'un objet

  (famille *set*xxxxx*)*

  ->Un objet sera immutable si il ne possède pas de mutateur

Méthodes : classification, un schéma

```
public class Exemple{
  private int x; // donnée d'instance
  ……
  // constructeur
  public Exemple(int x){……

  // accesseur
  public int getX(){ return this.x;}

// mutateur
  public void setX(int x){ this.x=x;}

……
```

Méthodes : une classification(2)

- **Constructeur**
- **Accesseur**
- **Mutateur**

- **Observateur**

  Méthodes à valeur booléenne

  (famille *boolean isxxxx)*

- **Producteur**

  Méthodes engendrant une nouvelle instance

  (famille *Classe operationxxxx)*

Méthodes : classification, un schéma

```
public class Exemple{
  private int x; // donnée d'instance
  ……
  // constructeur
  public Exemple(int x){……

  // observateur
  public boolean isPositive(){ return x>0;}

// producteur
  public Exemple minus(){ new Exemple(x-1);}

……
```

Objet et Partage[Liskov Sidebar2.2,page 22]

- An Object is ***mutable*** if its state can change. For example, arrays are mutable.
- An Object is ***immutable*** if its state never changes. For example, strings are immutable.
- An Object is ***shared*** by two variables if it can be accessed through either them.
- If a mutable object is shared by two variables, modifications made through one of the variables will be visible when the object is used by the other

La classe Object, racine de toute classe Java

- public class java.lang.**Object {**

- public boolean **equals(Object obj)** {....}
- public final native Class **getClass();**
- public native int **hashCode()** {....}
- **....**
- public String **toString()** {...}

- ...
- protected native Object **clone**() ...{...}
- protected void **finalize**() ...{...}
- **...**
- }

Surcharge(overloading) et Masquage(overriding)

Avec ou sans héritage

- **Surcharge : même nom et signature différente (overloading)**

avec héritage

- **Masquage : même nom et signature égale (overriding)**

La classe PolygoneRegulier re-visité

```
public class PolygoneRegulier  extends Object{
  private int nombreDeCotes;
  private int longueurDuCote;

  PolygoneRegulier  ( int nCotes, int longueur) {....}

    public boolean equals( Object obj ) {
       if( ! ( obj instanceof PolygoneRegulier ) ) return false;
       PolygoneRegulier poly  = ( PolygoneRegulier ) obj;
       return    poly.nombreDeCotes == nombreDeCotes  &&
                 poly.longueurDuCote == longueurDuCote;
    }

    public String   toString( ) {
       return "<" + nombreDeCotes + "," + longueurDuCote +
"">";
    }
    .....
}
```

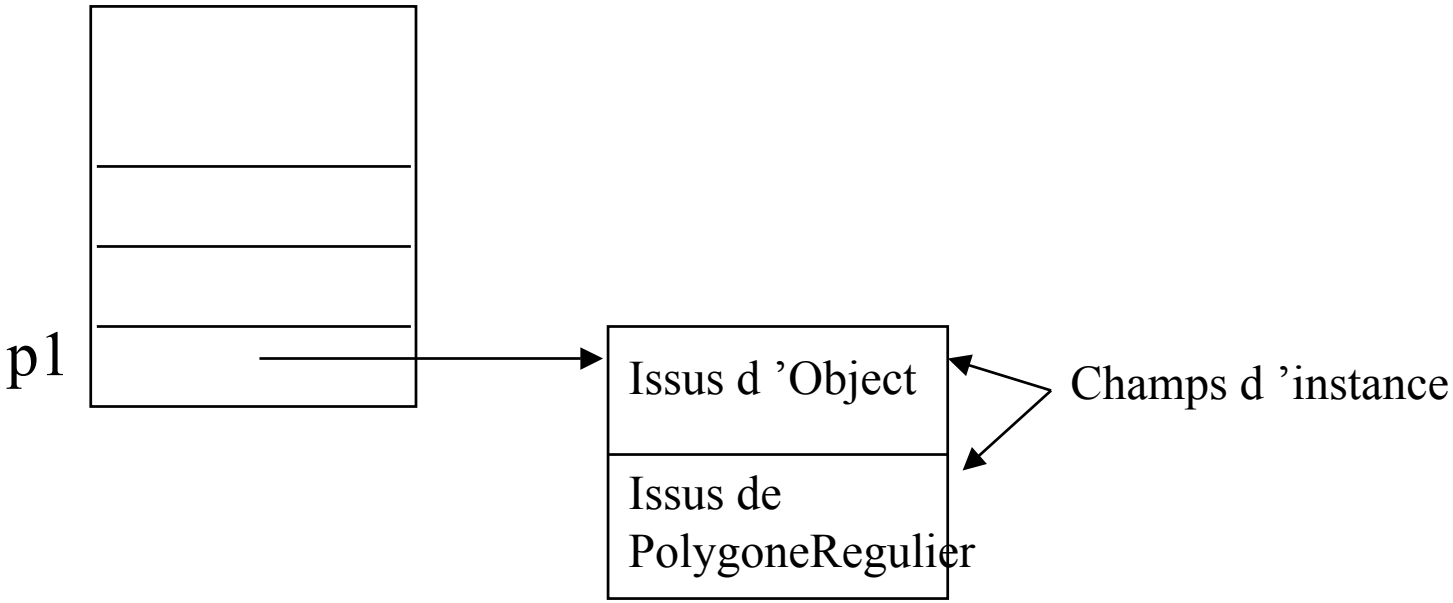
Exemple: utilisation

```
public class TestPolyReg{

    public static void main( String [ ] args ) {
        PolygoneRegulier p1 = new PolygoneRegulier ( 4, 100);
        PolygoneRegulier p2 = new PolygoneRegulier ( 5, 100);
        PolygoneRegulier p3 = new PolygoneRegulier ( 4, 100);

        System.out.println( "poly p1: " + p1.toString() );
        System.out.println( "poly p2: " + p2 );  // appel implicite de toString()
        System.out.println( "poly p3: " + p3 );
        if ( p1.equals( p2))
         System.out.println( "p1 == p2");
        else
          System.out.println( "p1 != p2");

        System.out.println( "p1 == p3 ? : " + p1.equals( p3 ) );
    }
}
```

Champs d'instance

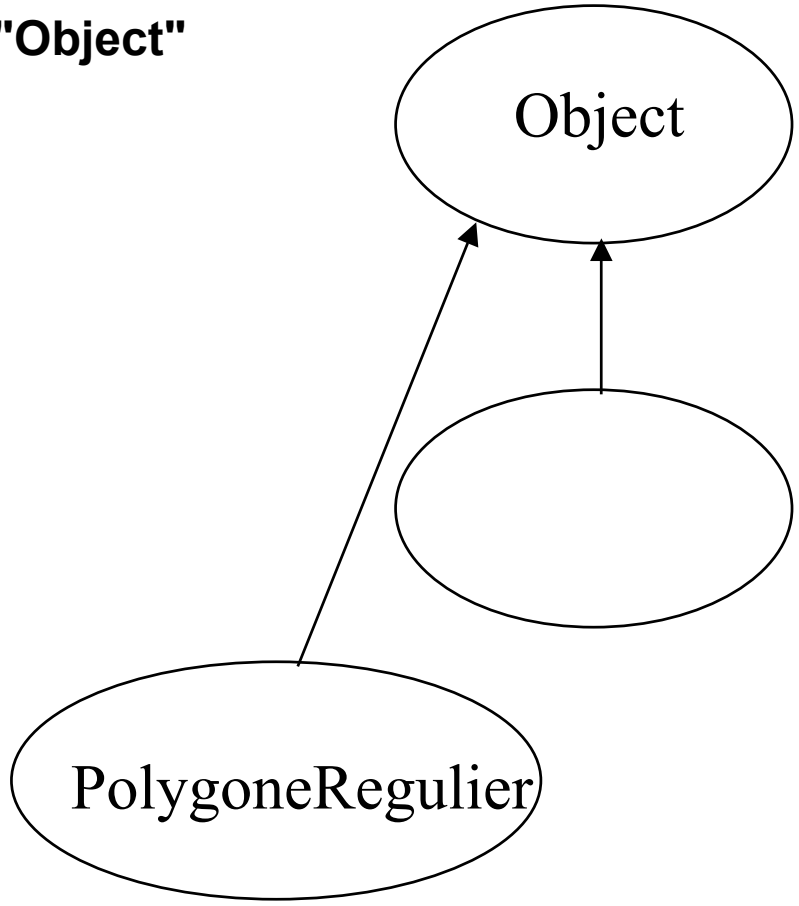
Class PolygoneRegulier extends Object{….}

p1 = new polygoneRegulier(4,100);



Affectation polymorphe

Toute instance de classe Java est un "Object"




```
Object obj = new Object();
PolygoneRegulier p = new
   PolygoneRegulier(5,100);

System.out.print( p.toString());


obj = p;

System.out.print( obj.toString());
```

Types et hierarchie[Liskov Sidebar2.4,page 27]

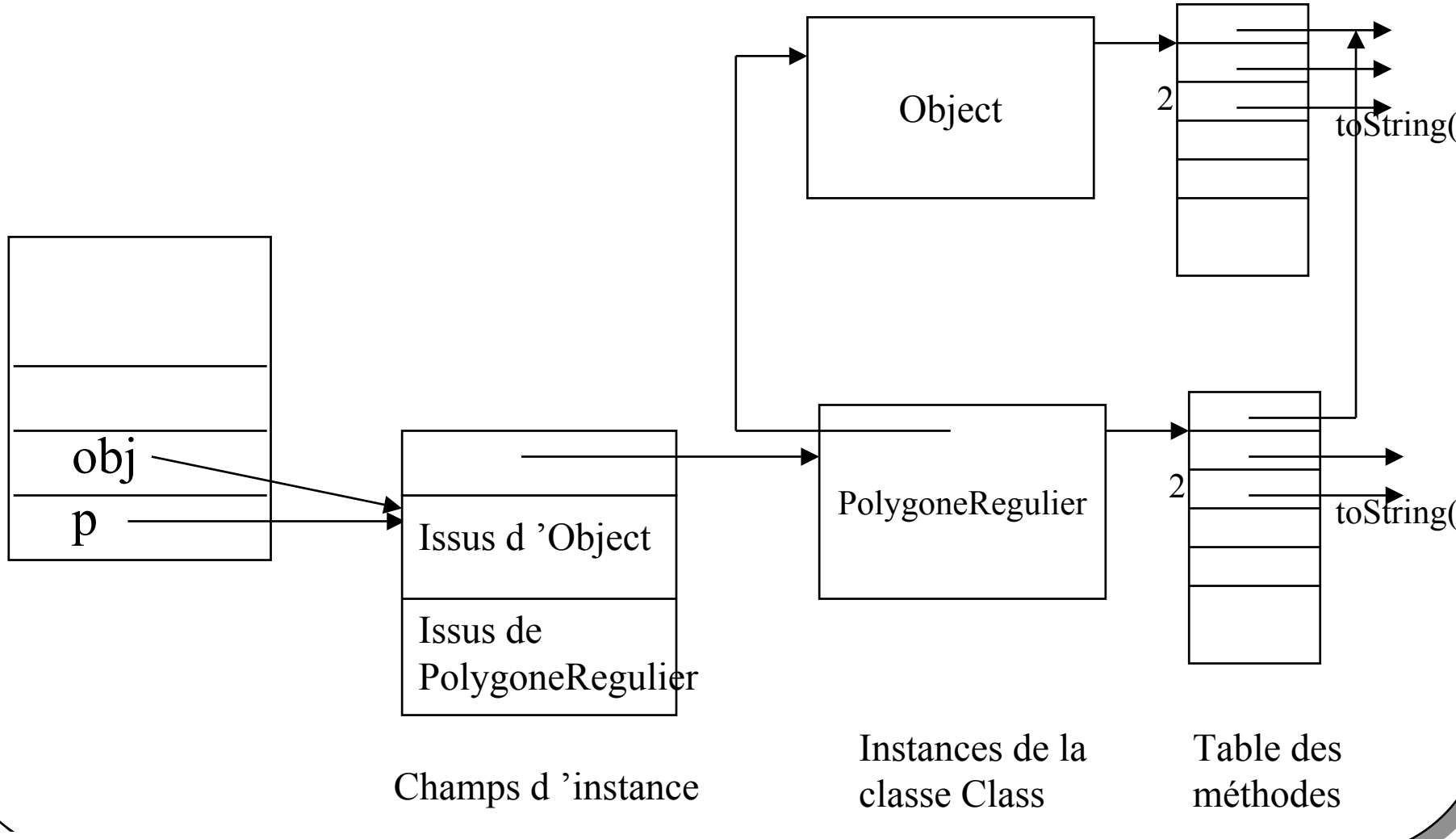
- Java supports *type* ***hierarchy***, in which one type can be the ***supertype*** of other types, which are its ***subtypes***. A subtype 's objects have all the methods defined by the supertype.
- All objects type are subtypes of **Object**, which is the top of the type hierarchy. **Object** defines a number of methods, including equals and toString. Every object is guaranteed to have these methods.
- The ***apparent type*** of a variable is the type understood by the compiler from information available in declarations. The ***actual type*** of an Object is its real type -> the type it receives when it is created.
- Java guarantees that the apparent type of any expression is a supertype of its actual type.

Ces notes de cours utilisent

- *type déclaré pour apparent type et*
- *type constaté pour actual type*

Sélection de la bonne méthode(1)

obj = p;
System.out.print(obj.toString()); // **Table[obj.classe constatée][2]**



Sélection de la bonne méthode(2)

- **Chaque instance référencée possède un lien sur sa classe**
  - **instance.getClass()**
- **Chaque descripteur de classe possède un lien sur la table des méthodes héritées comme masquées**
- **Le compilateur attribue un numéro à chaque méthode rencontrée**
- **Une méthode conserve le même numéro est tout au long de la hiérarchie de classes**
  - **obj.p() est transcrit par l 'appel de la 2 ème méthode de la table des méthodes associée à obj.**
  - **Quelque soit le type à l 'exécution de obj nous appelerons toujours la 2ème méthode de la table**
- **Comme contrainte importante**
  - **La signature doit être strictement identique entre les classes d 'un même graphe**
  - **boolean equals(Object) doit se trouver dans la classe PolygoneRegulier !!**
  - **(alors que boolean equals(PolygoneRegulier p) était plus naturel)**

Encapsulation

- **Une seule classe**
  - contrat avec le client
  - interface publique, en Java: outil javadoc
  - implémentation privée
  - Classes imbriquées
  - ==> Règles de visibilité

- **Plusieurs classes**
  - Paquetage : le répertoire courant est le paquetage par défaut
  - Paquetages utilisateurs
  - Paquetages prédéfinis
    - liés à la variable d'environnemement CLASSPATH

Règles de visibilité: public, private et protected

- **Tout est visible au sein de la même classe**

- **Visbilité entre classes**
  - **sous-classes du même paquetage**
  - **classes indépendantes du même paquetage**
  - **sous-classes dans différents paquetages**
  - **classes indépendantes de paquetages différents**

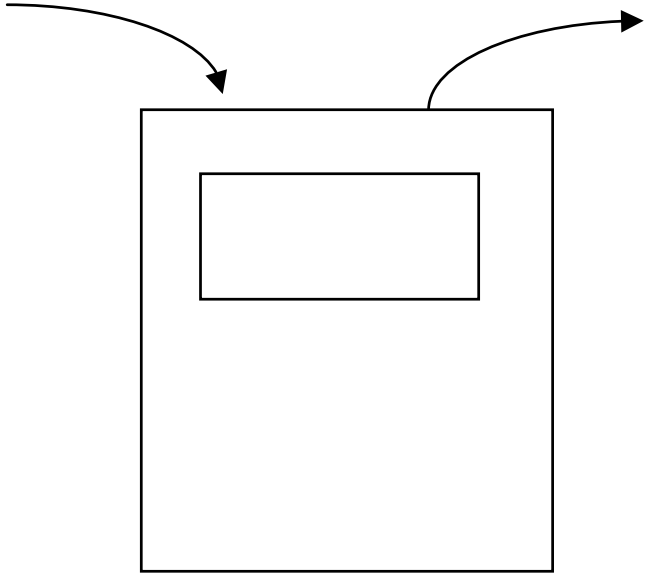
- **modifieurs d'accès**
  - **private**
  - **par défaut, sans modifieur**
  - **protected**
  - **public**

Règle d'accès

	private	défaut	**protected**	**public**
même classe	oui	oui	oui	oui
même paquetage et sous-classe	non	oui	oui	oui
même paquetage et classe indépendante	non	oui	oui	oui
paquetages différents et sous-classe	non	non	oui	oui
paquetages différents et classe indépendante	non	non	non	oui

Encapsulation classe imbriquée

- **Encapsulation effective**
- **Structuration des données en profondeur**
- **Contraintes de visibilité**



La classe imbriquée: exemple

```
public class PolygoneRegulier{
  private int nombreDeCotes;
  private int longueurDuCote;
  private Position pos;

  private class Position{
      private int x, int y;
      Position(int x, int y){
         this.x = x; this.y = y;
      }
      ....
   }

PolygoneRegulier( int nCotes, int longueur){
      .....
    pos = new Position(0,0);
      // ou pos = this.new Position(0,0);
}
```

La classe imbriquée statique

- public class PolygoneRegulier{
- private static int nombreDePolygones;

- **private static class Moyenne{**
- **private int x, int y;**
- // accès au nombreDePolygones (elle est statique)
- **}**

- PolygoneRegulier(int nCotes, int longueur){
- **.....**
- **moy = new** PolygoneRegulier **.Moyenne();**
- **// ou moy = new Moyenne();**
- }

Application Java : style d'écriture(3)

```
public class ApplicationJavaStyle3{

      public static void main(String[] args){
         new ApplicationJavaStyle3().leMain(args)
       }

          //usage de variables statiques (uniquement)
         // appels de méthodes statiques (uniquement)
        // ou apples de méthodes d 'instance
         proc(x,y);
      }

     public void proc( int i, int j){
       …
     }
      // variables d 'instance sont accessibles
}
```

La classe anonyme

- **Un usage : en paramètre d'une méthode**
- **exemple: l'opération "f" dépend de l'utilisateur,**

```
int executer(int x, Operation opr){
   return opr. f(x);
}
```

```
interface Operation{
  public int f(int x);
}
```

```
y = executer( 3, new Operation(){
                                public int f( int x){
                                     return x + 2;
                                }
                          });
// y == 5 !!!
```

syntaxe identique pour les classes

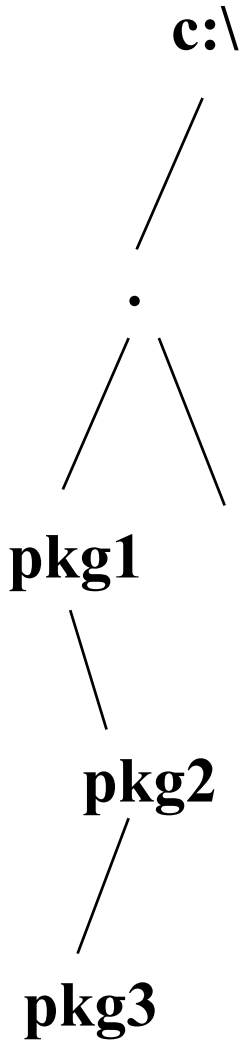
Package

- **Instructions**

  package ***pkg1[.pkg2[.pkg3];***

  import pkg1[.pkg2[.pkg3].(nomdeclasse|*);

- **Ils peuvent être regroupés dans un .ZIP et .JAR**



Paquetage java.

- de 8 paquetages en 1.0, 23 en 1.1
- de 211 classes en 1.0, 508 en 1.1, 1500 en 1.2, 1800 en 1.3 !!!!
- java.applet
  - classe Applet
  - interfaces AppletContext, AppletContext, AudioClip
- java.awt
  - Classes AppletButton, Color, Font, ...
  - interfaces LayoutManager, MenuContainer
  - java.awt.image
  - java.awt.peer
- java.io
  - .....
- java.lang
  - Classes Integer, Number, String, System, Thread,…..
  - interfaces Cloneable, Runnable
- java.net
- java.util
  - Classes Date, HashTable, Random, Vector, ...
  - interfaces Enumeration, Observer

L'interface

- **Protocole qu'une classe doit respecter**
- **Caractéristiques**
  - **Identiques aux classes, mais pas d'implémentation des méthodes**
  - **Toutes les variables d'instances doivent être "final", (constantes)**
- **Classes et interfaces**
  - **Une classe implémente une ou des interfaces**
  - **Les méthodes implémentées des interfaces doivent être publiques**
- **la clause *interface***

  visibilité interface *NomDeLInterface{*

  }
- **la clause implements**

  class NomDeLaClasse ***implements*** *nomDeLInterface, Interface1{*

  }

Interface et référence

- **une "variable Objet" peut utiliser une interface comme type**
- **une instance de classe implémentant cette interface peut lui être affectée**

```
interface I{
     void p();
}
class  A  implements  I {
    public void p() { .... }
}
class  B  implements  I {
    public void p(){ ....}
}
class Exemple {
     I i;
          i = new A();       i.p();       // i référence le contenu d 'une instance de type A

          i = new B();       i.p();       // i référence le contenu d 'une instance de type B
}
```

Exemple prédéfini : Interface Enumeration

- public interface java.util.Enumeration{
  public boolean **hasMoreElements**();
  public Object **nextElement**();
- }

- Toute classe implémentant cette interface doit générer une suite d'éléments, les appels successifs de la méthode ***nextElement*** retourne un à un les éléments de la suite

- exemple une instance **v** la classe java.util.Vector

```
for( Enumeration e = v.elements; e.hasMoreElements();){
    System.out.println( e.nextElement());
}
```

avec

```
public class java.util.Vector{
    ....
    public final Enumeration elements(){   }
```

Interface Enumeration

- **Parcours d ’une structure sans avoir à en connaître le contenu, : le pattern Iterator**

- **ajout d ’une nouvelle interface en 1.2 :**

```
public interface Iterator{
    boolean hasNext();
    Object next();
    void remove();
}
```

L'interface implémenté

- interface ObjetGraphique{
- public void dessiner();
- }

- class PolygoneRegulier implements ObjetGraphique{

```
public void dessiner(){
   }

....
```

Interface et « spécification »

- **interface PileSpec(**
- public void empiler(Object o);
- public Object depiler();
- public boolean EstVide ();
- public boolean estPleine();
- }

- **public class PileImpl** implements PileSpec {
- public void empiler(Object o){
- …..
- }
- public Object depiler(){
- …..
- }
- …..
- }

Déclaration d'interface imbriquée

```
public class Pile implements Pile.Spec {
        public interface Spec(
            public void empiler(Object o);
            public Object depiler();
            public boolean EstVide ();
            public boolean estPleine();
        }
public void empiler(Object o){
              …..
       }
        public Object depiler(){
              …..
       }
…..
}
```

Classe incomplète

- **Une classe avec l'implementation de certaines méthodes absente**
- **L'implémentation est laissée à la responsabilité des sous-classes**
- **elle n'est pas instanciable (aucune création d'instance)**

```
abstract class A{
  abstract void p();
  int i;                          // éventuellement données d'instance
  static int j;                                  // ou variables de classe
  void q(){
    // implémentation de q
  }
}
```

Classe incomplète: java.lang.Number

- abstract class Number ....{
- public abstract double doubleValue();
- public abstract float floatValue();
- public abstract int intValue();
- public abstract long longValue();
- }

- Dérivée par BigDecimal, BigInteger,Byte,Double,Float,Integer,Long,Short

Exceptions

- **Caractéristiques**
  - **Condition anormale d'exécution**
  - **instance de la classe "Throwable" ou une sous-classe**
  - **A l'occurrence de la condition, une instance est créée et levée par une méthode**
- **Hiérarchie de la classe "Throwable"**

```
Object
   |___Throwable
            |____Error
            | .....
            |____Exception
                     |___AWTException
                     |___ClassNotFoundException
                     |....
                     |___IOException
                     |___RunTimeException
                     |___Définie par l'utilisateur
```

Le bloc exception

- **try *{***
- instructions;
- instructions;
- instructions;
- ***}* catch*(ExceptionType1 e){***
- traitement de ce cas anormal de type ExceptionType1;
- ***}* catch*(ExceptionType2 e){***
- traitement de ce cas anormal de type ExceptionType2;
- **throw e;** *// l'exception est propagée*
- *//(vers le bloc try/catch) englobant*
- ***}* finally *(***
- *traitement de fin de bloc try ;*
- **}**

Le mot clé throws

- type NomDeMethode (Arguments) **throws** liste d'exceptions {
  corps de la méthode
- }

- **identifie une liste d'exceptions que la méthode est susceptible de lever**
- **l'appelant doit filtrer l'exception par la clause "catch" et/ou propager celle-ci (throws)**

- **RunTime Exceptions**
  Levée par la machine java
  Elles n'ont pas besoin d'être filtrée par l'utilisateur, un gestionnaire est installé par défaut par la machine

- **Exception définie par l'utilisateur**

```
class MonException extends Exception{
    ....
  }
```

Exemple d ’une structure de données : une liste

- Un interface et une classe d ’implémentation
- Une classe exception
- Une classe de tests

soient :

- public interface UnboundedBuffer{ ....}
- public class Queue implements UnboundedBuffer{ ....}
  class Node

- public class BufferEmptyException{ ....}

- public class TstQueue{ ....}

Une liste: interface UnboundedBuffer

- public interface UnboundedBuffer{
- public int count();
- public void put(Object x);
- public Object take() throws BufferEmptyException;
- }

Une liste: Queue.java (1)

```
public class Queue implements UnboundedBuffer {

  class Node {
      Object obj;
      Node next;
      Node(Object obj, Node next){
          this.obj  = obj;
          this.next = next;
    }
  }

  protected int size;
  protected Node head,tail;

Queue(){
    size = 0;
    head = tail = null;
  }
```

Une liste: Queue.java(2)

```
public int count(){  return size; }

 public void put(Object x) {
   if(size == 0){
     head = tail = this.new Node(x,null);
   }else{
     Node temp = this.new Node(x,null);
     tail.next = temp;
     tail = temp;
   }
   size++;
 }

 public Object take() throws BufferEmptyException {
   Object obj;
   if(size > 0){
     obj = head.obj;
     head = head.next;
     size--;
     return obj;
   }else{
     throw new BufferEmptyException();
   } }
```

Une liste: Queue.java(3)

```
public boolean empty(){
  return this.head == null;
}
```

```
public String toString(){
  String s = new String();
  java.util.Enumeration e = this.elements();
  while (e.hasMoreElements()){
    s = s + e.nextElement().toString();
  }
  return s;
}
```

Une liste: Queue.java(4)

```
• public java.util.Enumeration elements(){

•   class QueueEnumerator implements java.util.Enumeration{
•     private Queue queue;  private Node  current;

•     QueueEnumerator(Queue q){queue = q; current = q.head; }

•     public boolean hasMoreElements(){return current != null;}

•     public Object nextElement(){
•       if(current != null){
•         Object temp = current.obj;
•         current = current.next;
•         return temp;
•       }else{
•         return null;
•       }
•     }}
•   return new QueueEnumerator(this);
•  }
```

Une liste: BufferEmptyException

- public class BufferEmptyException extends Exception{}

Une liste: TstQueue

```
public class TstQueue{
  public static void main(String args[]) throws
BufferEmptyException {
    PolygoneRegulier p1 = new PolygoneRegulier(4,100);
    PolygoneRegulier p2 = new PolygoneRegulier(5,100);

     Queue q = new Queue();


     for(int i=3;i<8;i++){
       q.put(new PolygoneRegulier(i,i*10));
     }


    System.out.println(q.toString());
}
}
```

La classe Polygone

```
public class PolygoneRegulier{
  private int nombreDeCotes;
  private int longueurDuCote;

PolygoneRegulier( int nCotes, int longueur){
    nombreDeCotes = nCotes;
    longueurDuCote = longueur;
  }
  int perimetre(){
    return nombreDeCotes * longueurDuCote;
  }
  int surface(){
    return (int) (1.0/4 * (nombreDeCotes * Math.pow(longueurDuCote,2.0) *
                    cotg(Math.PI / nombreDeCotes)));
  }
  private static double cotg(double x){
    return Math.cos(x) / Math.sin(x);
  }
}
```

Exercice : une pile

- **Développer la traditionnelle structure de données de type pile (dernier entré premier sorti), les éléments sont de type "Object"**

  (Préferer une structure interne constituée d'un tableau et d'un indice)

  avec

  interface

  ce que toute pile doit respecter,

  implémentation de la class Pile

  (ne pas oublier toString(), equals et elements)

  exceptions : PileVideException, PilePleineException

  class TestPile

  (les éléments pourraient être des polygones réguliers !)

  documentation par javadoc

Exercice : suite

- La classe "*ArithmeticExpression*" propose la conversion d'une expression infixée en une expression postfixée, la grammaire de l'expression arithmétique est inspirée de la grammaire du langage Java, les nombres sont compris entre 0 et 9.
- **exemple : 3+2*5 est convertie en 325*+**

  □
- On demande de compléter la classe "*ArithmeticExpression*" en implémentant la méthode ***evaluate***, l'évaluation d'une expression postfixée utilise une instance de la classe *pile*

  325*+ doit engendrer

```
empiler(3);
empiler(2);
empiler(5);
empiler( depiler()  * depiler());
empiler( depiler()  + depiler());
```

  fichiers utilisés :

  ArithmeticExpression.java

  ArithmeticExpressionFormatException.java

  TestArithmeticExpression.java