
NFP121, Cnam/Paris

Cours 2

jean-michel Douin, douin au cnam point fr
version du 01 Octobre 2019

Notes de cours java : le langage java, une approche impérative

Sommaire

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

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

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 >

Démonstration

Exemple, bis: la classe des polygones réguliers

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

données
d'instance



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

méthodes



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

```
    public 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); }  
}
```

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 déclarée
- à laquelle se trouvent les champs d'instance et d'autres informations, (cette référence ne peut être manipulée)

- **Opérateur "."**

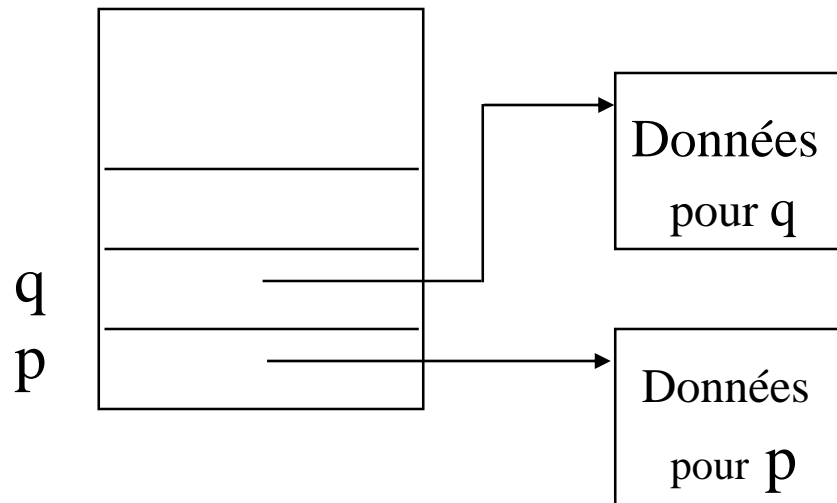
- appels de méthodes (si accessibles)
- accès aux champs (si accessibles)

- **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

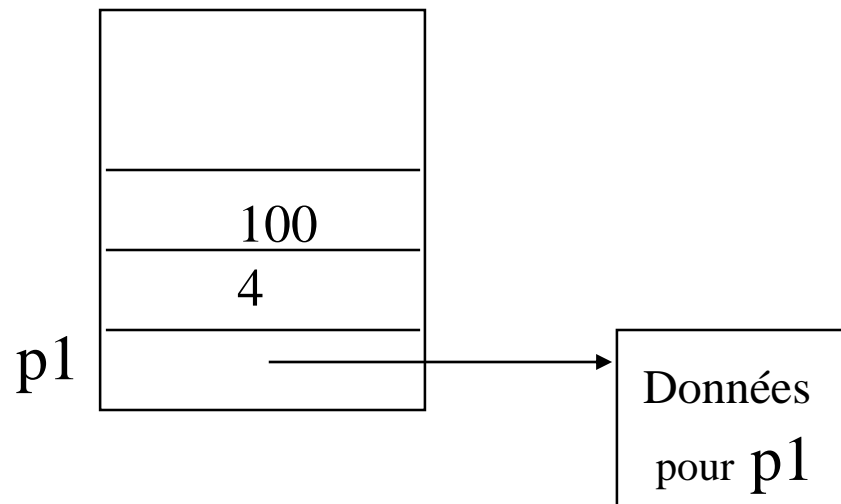
- `public class TestPolyReg{`
- `public static void main(String args[]){`
- `PolygoneRegulier p1 = new PolygoneRegulier();`
- `PolygoneRegulier p2 = new PolygoneRegulier();`
- `p1.initialiser(4,100);`
- `System.out.println(" surface de p1 = " + p1.surface());`
- `p2.initialiser(5,10);`
- `System.out.println(" perimetre de p2 = " + p2.perimetre());`
- `}`

création d'instance

appels de méthodes

Instances et appels de méthodes

`p1.initialiser(4,100);`



Pile d'exécution
avant l'appel de la
méthode initialiser

Initialisation des variables d'instance

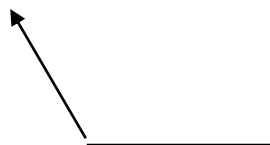
- **champs d'instance initialisés**
 - à 0 si de type entier, 0.0 si flottant, 0 si char, false si booléen
 - à null si référence d'objet
- **Variables locales aux méthodes**
 - elles ne sont pas initialisées, erreur à la compilation si utilisation avant affectation

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();



appel du constructeur
par défaut

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()`)

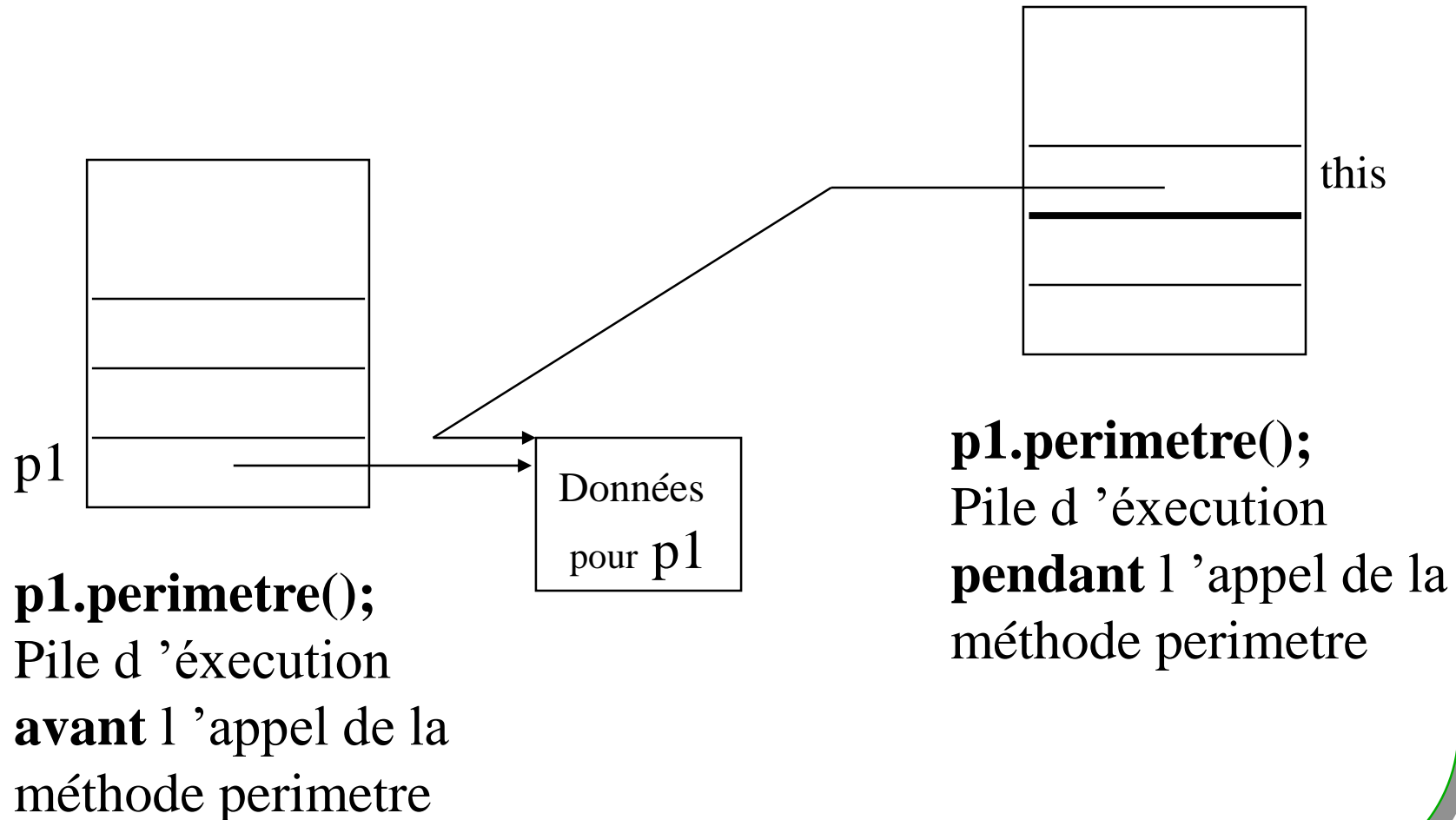
- **Garbage collection, Richard Jones and Rafael Lins. Wiley**
- **http://www.ukc.ac.uk/computer_science/Html/Jones/gc.html**

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;  
}
```



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){...};`
- `....`
- `}`

Démonstration suite

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;
```

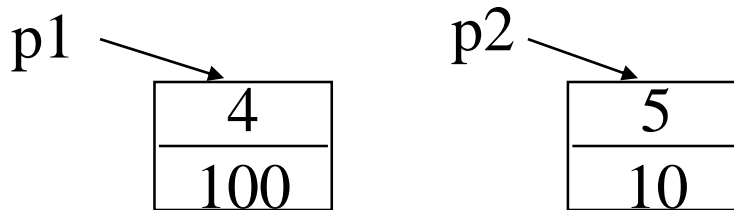
```
    public PolygoneRegulier ( int nCotes, int longueur) {  
        nombreDeCotes = nCotes;  
        longueurDuCote = longueur;  
    }
```

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

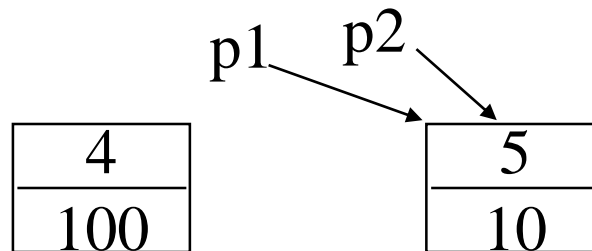
```
}
```

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)

- 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;
- public PolygoneRegulier (int nCotes, int longueur) {....}
-
- public String **toString**() { // redéfinition
- return "<" + nombreDeCotes + "," + longueurDuCote + ">";
- }
-
- }

Exemple: utilisation, démonstration

- `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()`
- `Object o = p1;`
- `System.out.println(" o: " + o);`
- `}`
- `}`

La classe PolygoneRegulier re-re-visité

- `public class PolygoneRegulier extends Object{`
- `private int nombreDeCotes;`
- `private int longueurDuCote;`
- `PolygoneRegulier (int nCotes, int longueur) {....}`
- `public boolean equals(PolygoneRegulier poly) { /**`
- `return poly.nombreDeCotes == nombreDeCotes &&`
- `poly.longueurDuCote == longueurDuCote;`
- `}`
- `public String toString() {`
- `return "<" + nombreDeCotes + "," + longueurDuCote + ">";`
- `}`
- `.....`
- `}`
- * une nouvelle définition de equals sur les polygones réguliers ...

Example: 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("p1 == p3 : " + p1.equals(p3));`
- `}`
- `}`
- **`p1 == p3 : true`**

– **Correct mais**

Exemple: utilisation, les hypothèses

- `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);`
- `Object o1 = p1;`
- `Object o3 = p3;`

// hypothèses

- `o1 doit se comporter comme p1, et o3 comme p3`

Exemple: utilisation, le résultat ???? !!!!

- Object o1 = p1;
- Object o3 = p3;
- // o1 doit se comporter comme p1, et o3 comme p3,

// existe une méthode equals de la classe Object -> la compilation réussit
public boolean equals(Object o);

- Or à l'exécution :

boolean b2 = o1.equals(o3); // b2 est faux // ??????

boolean b1 = o1.equals(p3); // b1 est faux // ??????

boolean b3= p1.equals(o3); // b3 est faux // ??????

boolean b4 = p1.equals(p3); // b4 est vrai // !!!!!!!

Démonstration

boolean b2 = o1.equals(o3); // b2 est faux // ??????

boolean b1 = o1.equals(p3); // b1 est faux // ??????

boolean b3= p1.equals(o3); // b3 est faux // ??????

equals correction

- `// public boolean equals(Object obj) { // ?`
- `public boolean equals(PolygoneRegulier poly) {`
- `return poly.nombreDeCotes == nombreDeCotes &&`
- `poly.longueurDuCote == longueurDuCote;`
- `}`
- Pourtant plus naturelle est une nouvelle méthode
- Pour `o1.equals(o3)`, c'est une redéfinition qu'il nous faut

Démonstration

La classe PolygoneRegulier re-re-visité

- `public class PolygoneRegulier extends Object{`
- `private int nombreDeCotes;`
- `private int longueurDuCote;`
- `PolygoneRegulier (int nCotes, int longueur) {...}`
- `@Override`
- `public boolean equals(Object obj) {` // ← redéfinition
- `if(! (obj instanceof PolygoneRegulier)) return false;` // ← discussion
- `PolygoneRegulier poly = (PolygoneRegulier) obj;`
- `return poly.nombreDeCotes == nombreDeCotes &&`
- `poly.longueurDuCote == longueurDuCote;`
- `}`
- `@Override`
- `public String toString() {`
- `return "<" + nombreDeCotes + "," + longueurDuCote + ">";`
- `}`
- `.....`
- `}`

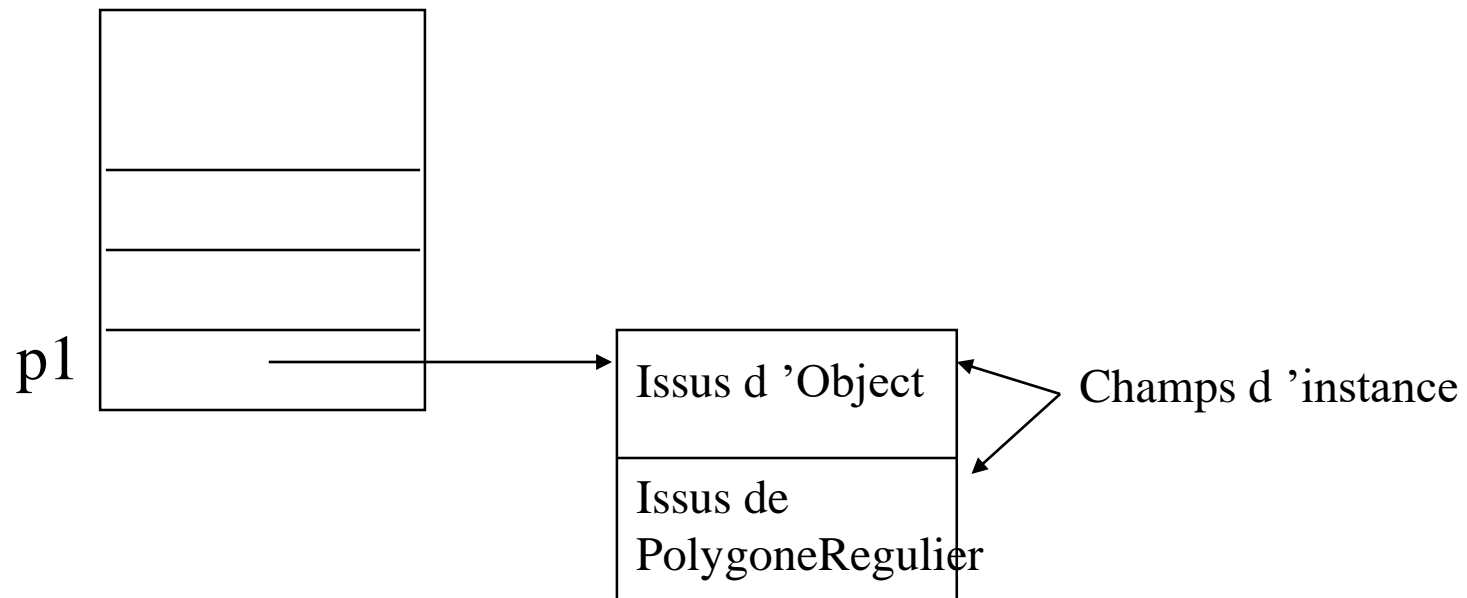
Démonstration-suite

- **Exactement la même signature**

Champs d'instance

Class PolygoneRegulier extends Object{....}

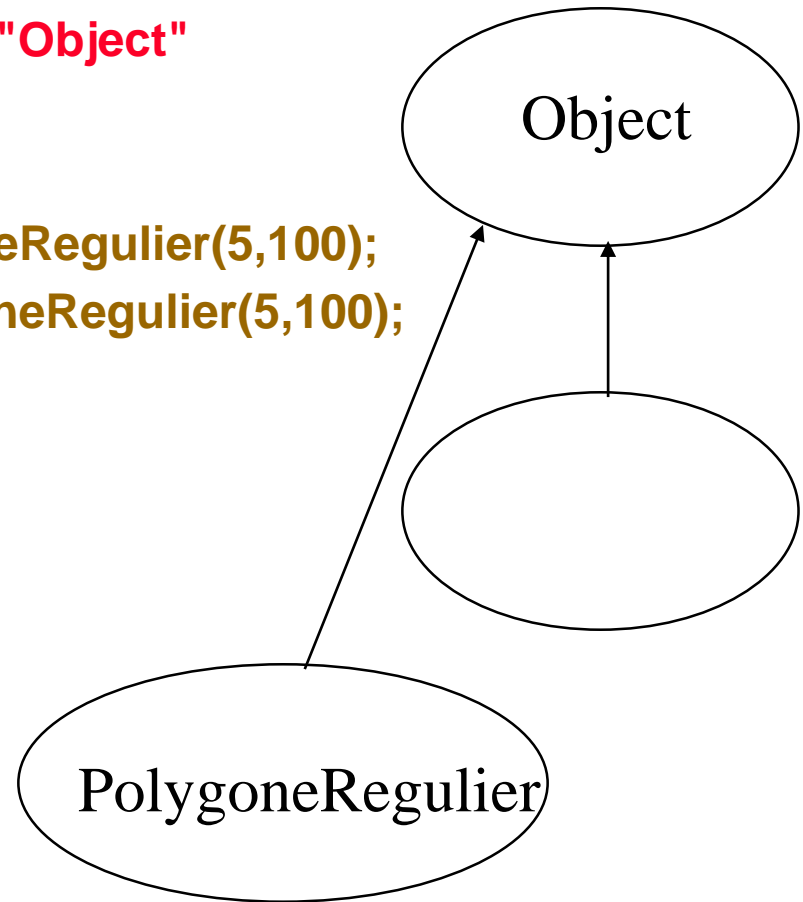
```
p1 = new polygoneRegulier(4,100);
```



Affectation polymorphe

Toute instance de classe Java est un "Object"

- `Object o = new Object();`
- `PolygoneRegulier p = new PolygoneRegulier(5,100);`
- `PolygoneRegulier p1 = new PolygoneRegulier(5,100);`
- `System.out.print(p.toString());`
- `o = p;`
- `System.out.print(o.toString());`
- `Object o1 = p1;`
- `boolean b = o1.equals(o);`



Types et hiérarchie[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.

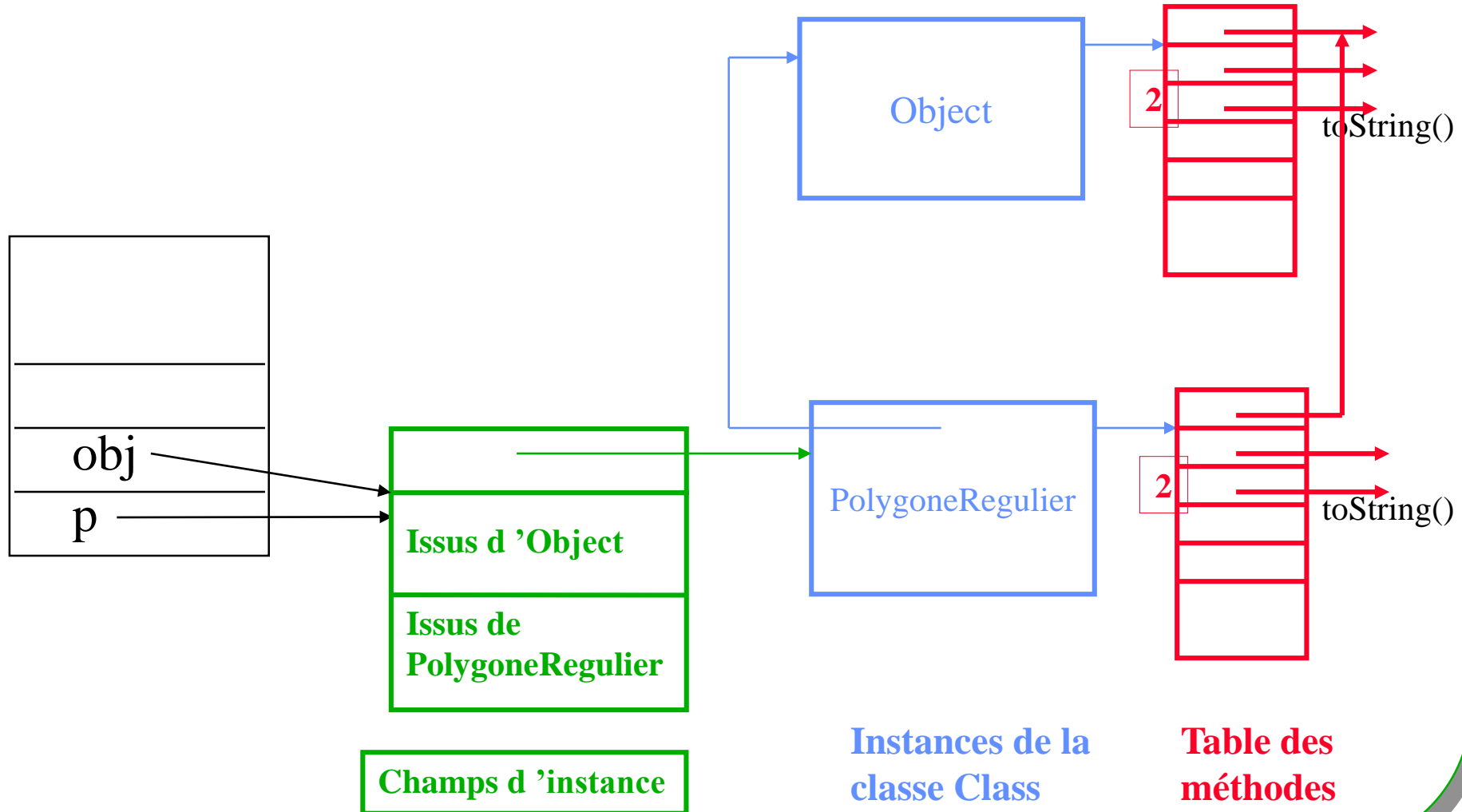
Ces notes de cours utilisent

- *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 tout au long de la hiérarchie de classes**
 - `obj.p()` est transcrit par l'appel de la 2ème méthode de la table des méthodes associée à `obj`.
 - Quelque soit le type à l'exécution de `obj` nous appellerons toujours la 2ème méthode de la table
- **Comme contrainte importante**
 - La signature doit être strictement identique entre les classes d'un même graphe d'héritage
 - *`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'environnement CLASSPATH**

Règles de visibilité:

public, private et protected

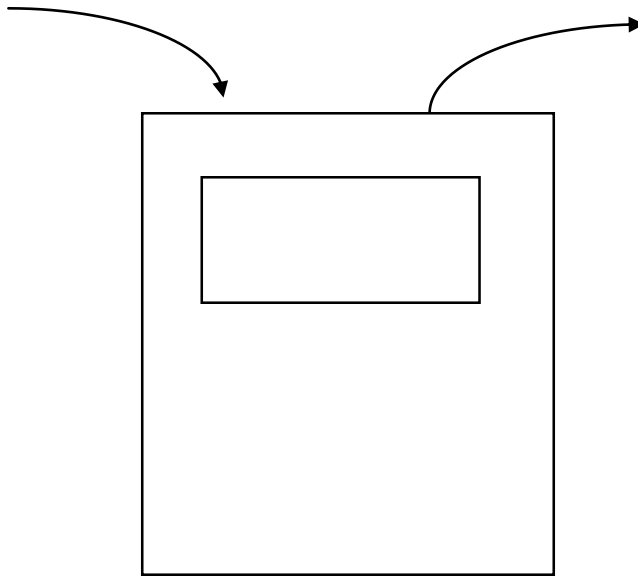
- **Tout est visible au sein de la même classe**
- **Visibilité entre classes**
 - sous-classes du même paquetage
 - classes indépendantes du même paquetage
 - sous-classes dans différents paquetages
 - classes indépendantes de paquetages différents
- **modificateurs d'accès**
 - private
 - par défaut, sans modificateur
 - 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**

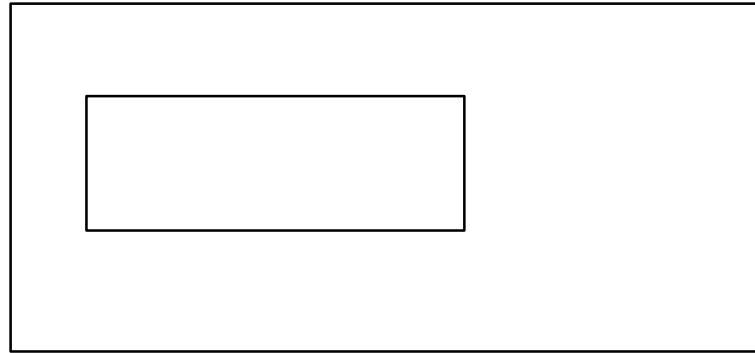


Quelques classes

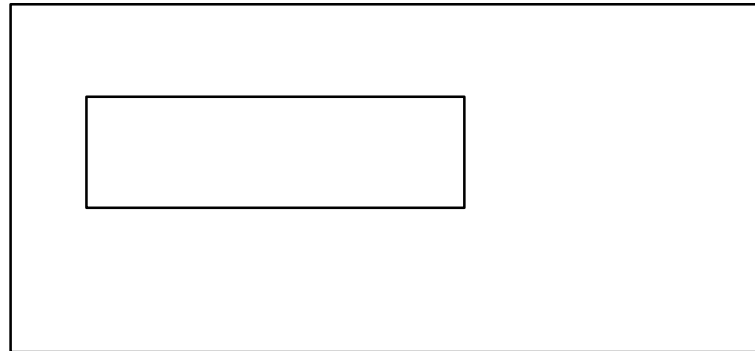
- Pour la syntaxe et pour l'exemple ...

- **Classe interne**

- Et membre



- Et statique



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();`**
- `}`

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);  
– }
```

```
class Operation{  
    public int f(int x){...}  
}
```

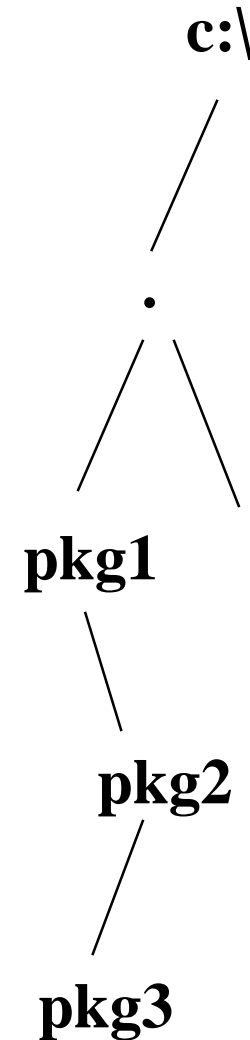
```
– int y = executer( 3, new Operation(){  
–                                     public int f( int x){  
–                                     return x + 2;  
–                                     }  
–                                     });  
– // y == 5 !!!
```

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, 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, première approche

- **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() { }`

Enumeration++ Interface Iterator

- ```
public interface java.util.Iterator<T>{
 – public boolean hasNext();
 – public T next();
 – public void remove();
}
```
- exemple une instance **v** la classe `java.util.Vector<Integer>`
  - `Iterator<Integer> it = v.iterator();`
  - **`while(it.hasNext()){`**  
    **`System.out.println(it.next());`**
  - **`}`**

avec

```
public class java.util.Vector<E>{

 public final Iterator<E> iterator(){ }
```

# 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{  
    ....  
}

# Démonstration

---

# Java quelques éléments de syntaxe

---

- **Corps d'une méthode**
- **types primitifs**
- **variables de type primitif**
- **types structurés**
- **Opérateurs**
- **Instructions**



# Types primitifs

---

- **entier**
  - **signés seulement**
  - type **byte** ( 8 bits), **short** ( 16 bits), **int** ( 32 bits), **long** ( 64 bits)
- **flottant**
  - **standard IEEE**
  - type **float**( 32 bits), **double** (64bits)
- **booléen**
  - type **boolean** (true,false)
- **caractère**
  - **unicode**,
  - type **char** (16 bits) <http://www.unicode.org>

# Variables de type primitif

---

- *type nom\_de\_la\_variable;*
- *type nom1,nom2,nom3;*
- *type nom\_de\_la\_variable = valeur;*
- **exemples :**
  - `int i;`
  - `int j = 0x55AA0000;`
  - `boolean succes = true;`
- **l'adresse d'une variable ne peut être déterminée**
- **le passage de paramètre est par valeur uniquement**

# Passage de paramètres par valeur uniquement

```
- static int plusGrand(int x, int y, int z){
- ...
- }
```

```
- int a=2, b=3, c=4;
- plusGrand(a,b,4);
```

|   |   |
|---|---|
|   |   |
|   |   |
|   |   |
| c | 4 |
| b | 3 |
| a | 2 |

Appel de plusGrand(a,b,4);



|   |   |
|---|---|
|   |   |
|   |   |
| z | 4 |
| y | 3 |
| x | 2 |
|   |   |
| c | 4 |
| b | 3 |
| a | 2 |

# Type entier et Changement de type

- **Automatique**

- *si la taille du type destinataire est supérieure*
- `byte a,b,c;`
- `int d = a+b/c;`

- **Explicite**

- `byte b = (byte) 500;`
- `b = (byte)( b * 2);`                      *// b \* 2 promue en int*
- *par défaut la constante numérique est de type int,*
- *suffixe L pour obtenir une constante de type long*    `40L`
- *par défaut la constante flottante est de type double,*
- *suffixe F pour obtenir une constante de type float*    `40.0F`

# Conversions implicites

---

- **Automatique**

- *si la taille du type destinataire est supérieure ....*

- byte                   ->     short,int,long,float,double

- short                 ->     int, long, float, double

- char                  ->     int, long, float, double

- int                   ->     long, float, double

- long                  ->     float, double

- float                 ->     double

# Application Java, un exemple

```
public class Application{
```

Nom de la classe -->  
fichier Application.java

```
public static void main(String args[]){
```

Point d'entrée unique  
la procédure "main"  
args : les paramètres de la  
ligne de commande

```
int i = 5;
```

```
i = i * 2;
```

variable locale à "main"

```
System.out.print(" i est égal à ");
System.out.println(i);
```

Instructions

affichage

```
}
```

```
}
```

# La classe Conversions : Conversions.java

---

```
public class Conversions{

 public static void main(String args[]){
 byte b;short s;char c;int i;long l;float f;double d;

 b=(byte) 0; s = b; i = b; l = b; f = b; d = b;

 i = s; l = s; f = s; d = s;

 i = c; l = c; f = c; d = c;

 l = i; f = i; d = i;

 f = l; d = l;

 d = f;
 }
}
```

# Type caractère

---

- Java utilise le codage Unicode
- représenté par 16 bits
- \u0020 à \u007E code ASCII, Latin-1
  - \u00AE ©
  - \u00BD / la barre de fraction ...
- \u0000 à \u1FFF zone alphabets
  - ....
  - \u0370 à \u03FF alphabet grec
  - .....
- <http://www.unicode.org>



# Opérateurs

- **Arithmétiques**

- +, -, \*, /, %, ++, +=, -=, /=, %=, --,

*Syntaxe C*

- **Binaires**

- ~, &, |, ^, &=, |=, ^=

- >>, >>>, <<, >>=, >>>=, <<=

- **Relationnels**

- ==, !=, >, <, >=, <=

*Syntaxe C*

- **Booléens**

- &, |, ^, &=, |=, ^=, ==, !=, !, ?:

- ||, &&

# Opérateurs booléens et court-circuits, exemple

```
• 1 public class Div0{
• 2 public static void main(String args[]){
• 3 int den = 0, num = 1;
• 4 boolean b;
• 5
• 6 System.out.println("den == " + den);
• 7
• 8 b = (den != 0 && num / den > 10);
• 9
• 10 b = (den != 0 & num / den > 10);
• 11 }
• 12 }
```

*Exception in thread "main" java.lang.ArithmeticException :  
/ by zero at Div0.main(Div0.java:10)*

# Précédence des opérateurs

|   |    |     |    |    |  |
|---|----|-----|----|----|--|
| + | () | []  | .  |    |  |
|   | ++ | --  | ~  | !  |  |
|   | *  | /   | %  |    |  |
|   | +  | -   |    |    |  |
|   | >> | >>> | << |    |  |
|   | >  | >=  | <  | <= |  |
|   | == | !=  |    |    |  |
|   | &  |     |    |    |  |
|   | ^  |     |    |    |  |
|   |    |     |    |    |  |
|   | && |     |    |    |  |
|   |    |     |    |    |  |
|   | ?: |     |    |    |  |
| - | =  | op= |    |    |  |

- `int a = 1, b = 1, c=2;`
- `int x = a | 4 + c >> b & 7 | b >> a % 3; // ??? Que vaut x ???`

# Type structuré : tableau

---

- **Déclarations de tableaux**

- `int[] mois; // mois est affecté à null ....`
- ou `int[] mois = new int[12];`
- ou `int[] mois={31,28,31,30,31,30,31,31,30,31,30,31};`

- **Déclarations de tableaux à plusieurs dimensions**

- `double [][] m= new double [4][4];`

- **Accès aux éléments**

- le premier élément est indexé en 0
- vérification à l'exécution des bornes, levée d'exception

- **En paramètre**

- la variable de type tableau est une référence,
- le passage par valeur de Java ne peut que transmettre la référence sur le tableau
- 2 syntaxes autorisées : `int mois[]` ou `int[] mois;` la seconde est préférée !, la première est devenue ancestrale ...

# Passage de paramètres par valeur uniquement

```
- static void trier(int[] tab){
- tab[0] = 8; // --> t[0] == 8;
- }
```

```
- int[] t = {2,4,3,1,5};
- trier(t);
```

Un extrait  
de la pile  
d'exécution

**t**



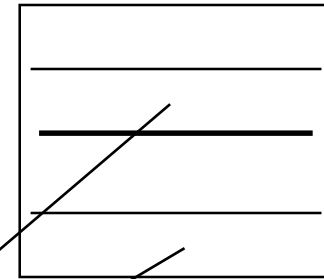
t[4]  
t[3]  
t[2]  
t[1]  
t[0]

|   |
|---|
| 5 |
| 1 |
| 3 |
| 4 |
| 2 |

**Appel de trier(t);**

**tab**

**t**



# Instructions de contrôle

---

- **branchement**
  - if else, break, switch, return // syntaxe C
- **Itération**
  - while, do-while, for, ' ', continue // syntaxe C
- **Exceptions**
  - try catch finally, throw

# Instruction de branchement, *if else*

- ***if ( expression-booleenne) instructions1; [else instructions2]***

```
public class IfElse{
 public static void main(String[] args){
 int mois = 4;
 String saison;

 if (mois == 12 || mois == 1 || mois == 2){
 saison = "hiver";
 } else if (mois == 3 || mois == 4 || mois == 5){
 saison = "printemps";
 } else if (mois == 6 || mois == 7 || mois == 8){
 saison = "ete";
 } else if (mois == 9 || mois == 10 || mois == 11){
 saison = "automne";
 } else {
 saison = "";
 }
 System.out.println("Avril est au " + saison + ".");
 }
}
```

# if (false) : compilation conditionnelle

---

```
public class IfElseDebug{
 public static final boolean DEBUG = false;

 public static void main(String[] args){
 int mois = 4;

 if (DEBUG) System.out.println(" mois = " + mois);
 else mois++;
 }
}
```

DOS> javap -c IfElseDebug

```
Method void main (java.lang.String[])
0 iconst_4 // int mois = 4;
1 istore_1
2 iinc 1 1 // mois++;
5 return
```



# Instructions de branchement, *switch case*

---

- ***switch ( expression) {***
- ***case value1 :***
- ***break;***
- ***case value2 :***
- ***.....***
- ***case value3 :***
- ***break;***
- ***case valueN :***
- ***break;***
- ***default :***
- ***}***

# switch case, exemple

---

```
public class SwitchSaison{
 public static void main(String[] args){
 int mois = 4; String saison;

 switch (mois){
 case 12: case 1: case 2:
 saison = "hiver";
 break;
 case 3: case 4: case 5:
 saison = "printemps";
 break;
 case 6: case 7: case 8:
 saison = "ete"; break;
 case 9: case 10: case 11:
 saison = "automne"; break;
 default:
 saison = "";
 }
 System.out.println("Avril est au " + saison + ".");
 }
}
```

# Itérations

---

- **while** ( *expression* ) {
- *instructions*
- }
  
- **for** (*initialisation; terminaison; itération*) *instructions;*
- $\Longleftrightarrow$
- *initialisation;*
- **while** (*terminaison*){
- *instructions;*
- *itération;*
- }

# Itération, *for( ; ; .),* exemple

---

```
public class Mois{
 public static void main(String[] args){

 String[] mois={"janvier","fevrier","mars","avril","mai","juin",
 "juillet","aout","septembre","octobre","novembre","decembre"};
 int[] jours={31,28,31,30,31,30,31,32,30,31,30,31};
 String printemps = "printemps";
 String ete = "ete";
 String automne = "automne";
 String hiver = "hiver";
 String[] saisons={ hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
 automne,automne,automne,hiver};

 for(int m = 0; m < 12; m++){
 System.out.println(mois[m] + " est au/en " +saisons[m] + " avec " +
 jours[m] + " jours.");
 }
 }
}
```

# break, continue

- **break;**     *fin du bloc en cours*
- **continue;**     *poursuite immédiate de l'itération*

- for ( .....){
- → .....
- **continue;**
- .....

- **etiquette :** for(.....){
- for ( .....){
- .....
- **continue etiquette;**
- .....
- }
- }
- }

# Les exceptions: présentation

---

- Condition **anormale** d'exécution d'un programme

```
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 ;
}
```

# Les exceptions : exemple

---

Soit l'instruction suivante :

```
m = Integer.parseInt(args[0]);
```

Au moins deux erreurs possibles :

1) `args[0]` n'existe pas

-> **`ArrayIndexOutOfBoundsException`**

2) le format de '`args[0]`' n'est pas celui d'un nombre

-> **`NumberFormatException`**

# Les exceptions : exemple

```
public class MoisException{
 public static void main(String[] args){
 String[] mois={"janvier", "fevrier", "mars", "avril", "mai", "juin",
 "juillet", "aout", "septembre", "octobre", "novembre", "decembre"};
 int[] jours={31,28,31,30,31,30,31,32,30,31,30,31};
 String printemps = "printemps"; String ete = "ete";
 String automne = "automne"; String hiver = "hiver";
 String[] saisons={hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
 automne,automne,automne,hiver};

 int m;
 try{
 m = Integer.parseInt(args[0]) -1;
 System.out.println(mois[m] + " est au/en " +saisons[m] + " avec " + jours[m] + " j.");
 }catch(ArrayIndexOutOfBoundsException e){
 System.out.println("usage : DOS>java MoisException unEntier[1..12]");
 }catch(NumberFormatException e){
 System.out.println("exception " + e + " levee");
 }finally{
 System.out.println("fin du bloc try ");
 }
 }
}
```



# L'exemple initial avec une division par zéro

```
1 public class Div0{
2 public static void main(String args[]){
3 int den = 0, num = 1;
4 boolean b;
5
6 System.out.println("den == " + den);
7
8 b = (den != 0 && num / den > 10);
9
10 b = (den != 0 & num / den > 10);
11 }
12}
```

// Il existe un donc un bloc try/catch prédéfini  
// interne à la machine virtuelle

*try{*

*Div0.main({""});*

*}catch (RuntimeException e) {*

*System.err.println(e);*

*}*

# Les exceptions sont des classes

```
java.lang.Object
|
+--java.lang.Throwable
 |
 +--java.lang.Exception
 |
 +--java.lang.RuntimeException
 |
 +--java.lang.IndexOutOfBoundsException
 |
 +--java.lang.ArrayIndexOutOfBoundsException
```

```
+--java.lang.RuntimeException
 ArithmeticException, ArrayStoreException, CannotRedoException,
 CannotUndoException, ClassCastException, CMMException,
 ConcurrentModificationException, EmptyStackException, IllegalArgumentException,
 IllegalMonitorStateException, IllegalPathStateException, IllegalStateException,
 ImagingOpException, IndexOutOfBoundsException, MissingResourceException,
 NegativeArraySizeException, NoSuchElementException, NullPointerException,
 ProfileDataException, ProviderException, RasterFormatException, SecurityException,
 SystemException, UnsupportedOperationException
```

# if(DEBUG) + throw new RuntimeException

---

```
public class IfElseDebug{
 public static final boolean DEBUG = false;

 public static void main(String[] args){
 int mois = 4;

 if (DEBUG) assert(mois==4);
 else mois++;
 }

 public static void assert(boolean b){
 if (!(b)) throw new RuntimeException("assertion fausse ") ;
 }
}
```

```
DOS> javap -c IfElseDebug
Method void main (java.lang.String[])
0 iconst_1
1 istore_1
2 iinc 1 1
5 return
```

# NullPointerException

```
- static void trier(int[] tab){
- // levée d'une exception instance de la classe
- // NullPointerException,
- }

-
- int[] t; /** t == null; */
- trier(t); // filtrée par la machine

-
- _____
- _____
- ou bien
- int[] t;
- try{
- trier(t);
- }catch(NullPointerException e){}
```

# Classe : Syntaxe, champs et méthodes "statiques"

---

- **public class** NomDeClasse{
  - **static** type variable1DeClasse;
  - **static** type variable2DeClasse;
  - **public static** type variableNDeClasse;
  - **static** type nom1DeMethodeDeClasse( listeDeParametres) {
  - }
  - **static** type nom2DeMethodeDeClasse( listeDeParametres) {
  - }
  - **static** type nomNDeMethodeDeClasse( listeDeParametres) {
  - }
  - **public static void** main(String args []){ } // ....
- }

*Notion Variables et méthodes globales*

# Accès aux champs de classe

---

- Le mot clé **static**
- **accès en préfixant par le nom de la classe**
  - **exemple la classe prédéfinie "Integer"**
  - package java.lang;
  - public class Integer ... {
  - public **static** final int MAX\_VALUE= ...;
  - ...
  - public **static** int parseInt(String s){ ...}
  - ...
  - }
  
  - **// accès :**
  - int m = **Integer**.parseInt("33");
- **Opérateur "."**
  - appels de méthodes de classe ( si elles sont accessibles)
  - accès aux champs de classe ( si accessibles)

# Variables et méthodes de classe

## Variables de classes, (notion de variables globales)

```
public class MoisBis{
 static String[]
 mois={"janvier","fevrier","mars","avril","mai","juin",

 "juillet","aout","septembre","octobre","novembre","decembre"};
 static int[] jours={31,28,31,30,31,30,31,32,30,31,30,31};
 static String printemps = "printemps";
 static String ete = "ete";
 static String automne = "automne";
 private static String hiver = "hiver";
 public static String[]
 saisons={hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
 automne,automne,automne,hiver};
 public static void main(String[] args){
 for(int m = 0; m < 12; m++){
 System.out.println(mois[m] + " est au/en " +saisons[m] + "
avec " +jours[m] + " jours.");
 }
 }
}
```

# Définition d'une Classe

---

- **modificateurs d'accès aux variables et méthodes**
  - accès : `public`, `private`, `protected` et « sans »
  - `final`
  - `static`
- **Déclararation et implémentation**
  - dans le même fichier `".java"` (`NomDeLaClasse.java`)
  - documentation par l'outil *javadoc* et `/** .... */`
  - une seule classe publique par fichier `".java"`
- **class Object**
  - toutes les classes héritent de la classe "Object"
  - méthodes de la classe Object
    - `String toString();`
    - `String getClass();`
    - `Object clone();`
    - `boolean equals(Object);`
    - `void finalize();`



# Le bloc static

## exécuté une seule fois au chargement de la classe

```
– public class MoisTer{
– static String[] mois ={"janvier","fevrier","mars","avril","mai","juin",
– "juillet","aout","septembre","octobre","novembre","decembre"};
– static int[] jours ={31,28,31,30,31,30,31,32,30,31,30,31};
– static String printemps,ete,automne,hiver;
– static String[] saisons ={ hiver,hiver,printemps,printemps,printemps,ete,ete,ete,
– automne,automne,automne,hiver};

– public static void main(String[] args){
– for(int m = 0; m < 12; m++){
– System.out.println(mois[m] + " est au/en " +saisons[m] + " avec " +
– jours[m] + " jours.");
– }
– }

– static{
– printemps = "printemps"; ete = "ete";
– automne = "automne"; hiver = "hiver";
– }}
```

# Application Java

---

- **Caractéristiques**
  - autonome, sans navigateur
  - une ou plusieurs classes
  - les classes sont chargées dynamiquement
- **Le point d'entrée *main***
  - l'application doit comporter une classe avec la méthode *main*
  - sa signature est `public static void main(String[] args)`
  - appelée par la commande `DOS> java` suivie du nom de la classe ayant implémentée la méthode *main*
- **Passage d'argument(s)**
  - sur la ligne de commande
  - `DOS> java NomDeClasse Octobre 30`
- **Gestion des paramètres dans l'application**
  - tous ces paramètres sont de "type" `String`
  - conversion en entier par exemple
  - `int x = Integer.parseInt(args[0]); // classe Integer`

# Application Java : style d'écriture(1)

---

```
public class ApplicationJavaStyle1{

 public static void main(String[] args){
 // variables locales
 int x, y;

 // appels de méthodes statiques (uniquement)
 proc(x,y);
 }

 public static void proc(int i, int j){
 ...
 }

}
```

# Application Java : style d'écriture(2)

---

```
public class ApplicationJavaStyle2{
 // Variables statiques
 static int x;
 static int y;

 public static void main(String[] args){

 //usage de variables statiques (uniquement)
 // appels de méthodes statiques (uniquement)
 proc(x,y);
 }

 public static void proc(int i, int j){
 ...
 }
}
```

# Package

- **Fonction**

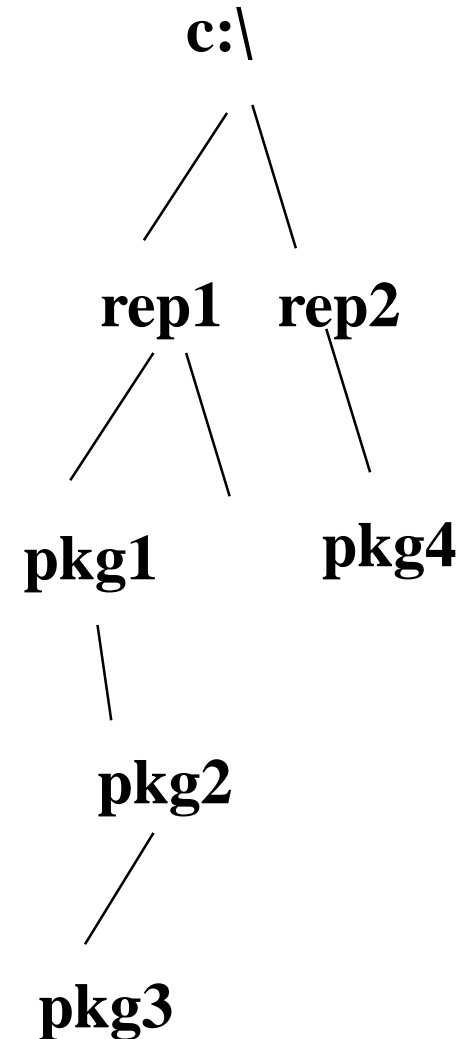
- Unité logique par famille de classes
- découpage hiérarchique des paquetages
- (ils doivent être importés explicitement sauf java.lang)

- **Buts**

- espace de noms
- restriction visibilité

- **Instructions**

- *package pkg1[.pkg2[.pkg3];*
- les noms sont en minuscules
- c'est la première instruction du source java
- *import pkg1[.pkg2[.pkg3].(nomdeclasse/\*);*
- liés aux options de la commande de compilation
- dos> javac -classpath .;c:\rep1;c:\rep2



# Paquetages prédéfinis

---

- **le paquetage `java.lang.*` est importé implicitement**
  - ce sont les interfaces : *Cloneable*, *Comparable*, *Runnable*
  - et les classes : `Boolean`, `Byte`, `Character`, `Class`, `ClassLoader`, `Compiler`,  
`Double`, `Float`, `InheritableThreadLocal`, `Long`, `Math`, `Number`,  
`Object`, `Package`, `Process`, `Runtime`, `RuntimePermission`,  
`SecurityManager`, `Short`, `StrictMath`, `String`, `StringBuffer`,  
`System`, `Thread`, `ThreadGroup`, `ThreadLocal`,  
`Throwable`, `Void`,
  - toutes les classes dérivées de `Exception`, `ArithmeticException`, ....
  - et celles de `Error`, `AbstractMethodError`, ....
- **`java.awt.*`   `java.io.*`   `java.util.*`   ....**

# Résumé

---

- **Types primitifs et type « Object » (et ses dérivés)**
- **Instructions analogue au langage C**
- **La classe et l'interface**
- **Variables de classe**
- **Application Java, methode de classe main**
- **Regroupement de classes en paquetage**

- **Le code source d'une liste**



# Exemple d'une structure de données : une liste

---

- Un interface et une classe d'implémentation
- Une classe exception
- Une classe de tests
  
- `public interface UnboundedBuffer{ ....}`
- `public class Queue implements UnboundedBuffer{ ....}`
  - `class Node`
  
- `public class BufferEmptyException{ ....}`
  
- `public class TestQueue{ ....}`

# 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 PolygoneRegulier

---

```
• 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);
• }
• }
```