

2i002 - Static

Vincent Guigue



STATIC ≠ POO

POO

- Un objet protège ses attributs
- Un objet possède des méthodes pour gérer ses attributs

Usage

- ① Création d'une instance
- ② Appel de méthode sur cette instance

Static

- Les attributs/méthodes *static* ne dépendent pas d'un objet
- Tous les objets d'une classe ont accès aux mêmes informations *static*

Usage

- ① Appel de méthode/attribut indépendamment des instances

USAGE DE STATIC

- **Attributs static (=variable de classe)** : partager des informations entre les classes
 - Compteurs

Combien d'instances de Point ont-elles été créées ?

Question non triviale avec les outils actuels !

- Liste des objets créés

Je voudrais accéder à n'importe quel point créé jusqu'ici...

- Constantes (cas particulier, cf plus loin)

π...

- **Méthodes static** : outils non reliés à une instance
 - Outils (opérations entre instances, opérations annexes)
 - Accesseur à un attribut static
 - Singleton

USAGE DE STATIC

- **Attributs static (=variable de classe) :** partager des informations entre les classes
 - Compteurs
 - Liste des objets créés
 - Constantes (cas particulier, cf plus loin)
- **Méthodes static** : outils non reliés à une instance
 - Outils (opérations entre instances, opérations annexes)

Ex : **cos**, une méthode n'utilisant aucun attribut, **utilisable directement, sans instantiation d'un objet de la classe Math**

- Accesseur à un attribut static
- Singleton

Programmation objet :

```
1 // Instantiation  
2 Point p = new Point(1,2);  
3  
4 // Invocation de méthode  
5 // SUR L'INSTANCE  
6 p.move(3, 3);  
7 p.toString();  
8 ...
```

Philosophie :

Les méthodes *accèdent / modifient* l'instance

Programmation objet :

```
1 // Instantiation  
2 Point p = new Point(1,2);  
3  
4 // Invocation de méthode  
5 // SUR L'INSTANCE  
6 p.move(3, 3);  
7 p.toString();  
8 ...
```

Philosophie :

Les méthodes *accèdent* /
modifient l'instance

Programmation static

```
1 // Pas d'instantiation de la classe  
2 // Appel directement sur la classe  
3 double pi = Math.PI;  
4  
5 // Pareil pour les méthodes  
6 double d = Math.cos(pi);
```

Philosophie :

- Pas d'instance, pas d'accès aux attributs
- Constante indépendante
- Méthode indépendante

Programmation objet :

```
1 // Instantiation  
2 Point p = new Point(1,2);  
3  
4 // Invocation de méthode  
5 // SUR L'INSTANCE  
6 p.move(3, 3);  
7 p.toString();  
8 ...
```

Philosophie :

Les méthodes *accèdent / modifient* l'instance

Programmation static

```
1 // Pas d'instantiation de la classe  
2 // Appel directement sur la classe  
3 double pi = Math.PI;  
4  
5 // Pareil pour les méthodes  
6 double d = Math.cos(pi);
```

Philosophie :

- Pas d'instance, pas d'accès aux attributs
- Constante indépendante
- Méthode indépendante

⇒ Essayons maintenant de mélanger les 2 philosophies pour faire des choses nouvelles

Combien d'instances de Point ont-elles été créées ?

Question non triviale avec les outils actuels !

Identifiant unique/comptage des instances

```
1 Point p1 = new Point(); // constructeur random  
2 Point p2 = p1;  
3 Point p3 = new Point(3,5);  
4 ...
```

- Combien d'instance ?
- Peut-on attribuer à chaque Point un identifiant unique (lié à son ordre de création) ?

COMPTAGE D'INSTANCES : SYNTAXE STANDARD

Forme standard

```
1 public class Point{  
2     private static int cpt = 0; // initialisation obligatoire ici  
3     private int id; // initialisation interdite ici (-> constr)  
4     private double x,y;  
5  
6     public Point(double x, double y){  
7         this.x = x; this.y = y;  
8         id = cpt++; // ou: id = cpt; cpt++;  
9     }  
}
```

- Chaque Point a :
 - un `x`, un `y`, un `id`
- Tous les Point partagent :
 - un compteur `cpt`

Le partage permet de raisonner sur des concepts qui dépassent UNE SEULE instance

COMPTAGE & PRÉSENTATION MÉMOIRE

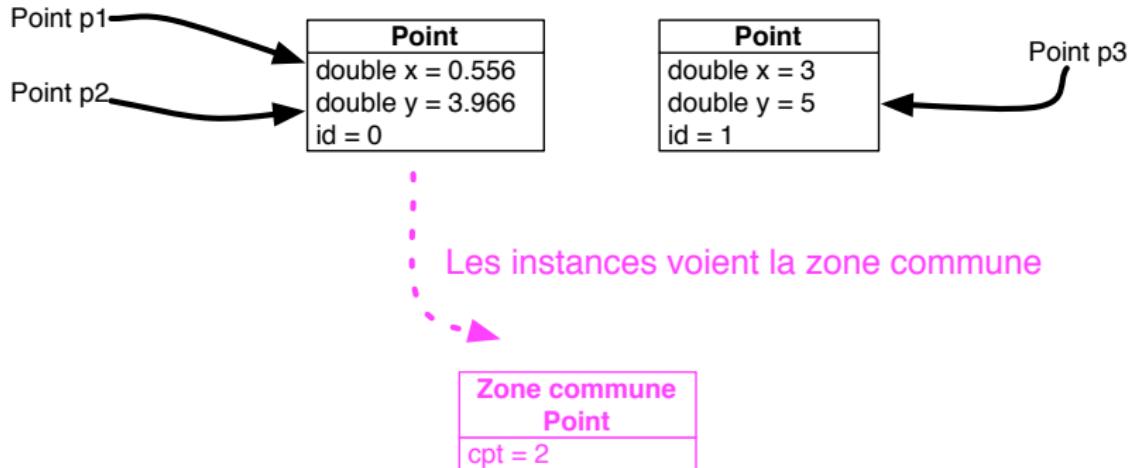
```
1 Point p1 = new Point();  
2 Point p2 = p1;  
3 Point p3 = new Point(3,5);
```

- Où se trouve l'id ?, Où se trouve le compteur ? (dans une représentation mémoire)

COMPTAGE & REPRÉSENTATION MÉMOIRE

```
1 Point p1 = new Point();  
2 Point p2 = p1;  
3 Point p3 = new Point(3,5);
```

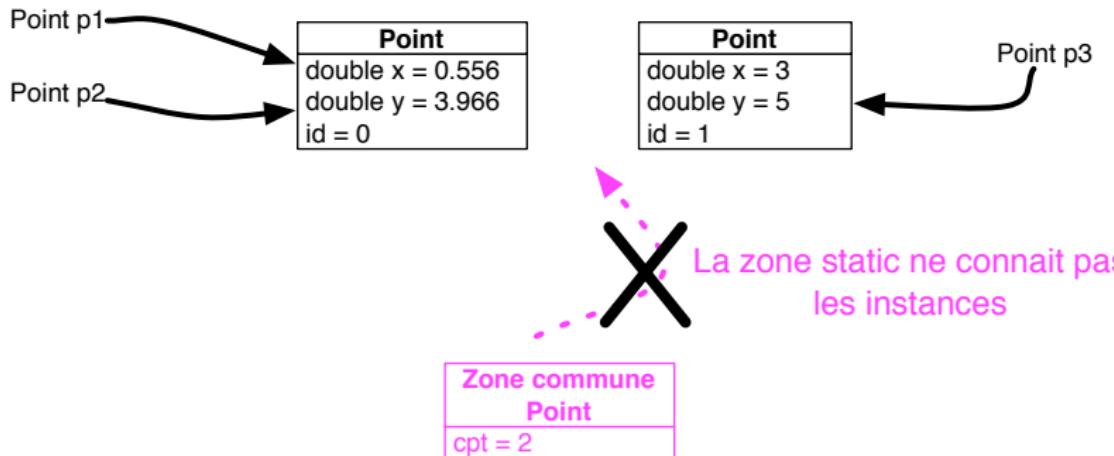
- Où se trouve l'id ?, Où se trouve le compteur ? (dans une représentation mémoire)



COMPTAGE & REPRÉSENTATION MÉMOIRE

```
1 Point p1 = new Point();  
2 Point p2 = p1;  
3 Point p3 = new Point(3,5);
```

- Où se trouve l'id ?, Où se trouve le compteur ? (dans une représentation mémoire)



Forme standard

```
1 public class Point{  
2     private static int cpt = 0; // initialisation obligatoire ici  
3     private int id; // initialisation interdite ici (-> constr)  
4     private double x,y;  
5  
6     public Point(double x, double y){  
7         this.x = x; this.y = y;  
8         id = cpt++; // ou: id = cpt; cpt++;  
9     }  
10  
11    // garantie de bonne gestion des id  
12    public Point(){  
13        this(Math.random()*10, Math.random()*10);  
14    }  
}
```

- Piège : attention aux constructeurs multiples
⇒ usage de `this()` très fortement conseillé pour passer toujours par le constructeur de référence et bien compter.

VARIABLES PARTAGÉES

- Variable de classe/static = partage d'information

Exemples :

- Combien d'instances ont été créées jusqu'ici ? (compteur)
- Modélisation bancaire : une banque, plusieurs agences, toutes les agences voient/modifient les l'ensemble des comptes...
- Maillage CAO : chaque triangle connaît tout le maillage (ensemble des instances créées) ⇒ interaction avec les voisins (alternative à une liste chainée)

VARIABLES PARTAGÉES

- Variable de classe/static = partage d'information

Exemples :

- Combien d'instances ont été créées jusqu'ici ? (compteur)
- Modélisation bancaire : une banque, plusieurs agences, toutes les agences voient/modifient les l'ensemble des comptes...
- Maillage CAO : chaque triangle connaît tout le maillage (ensemble des instances créées) ⇒ interaction avec les voisins (alternative à une liste chaînée)

Toujours vérifier qu'une variable static **ne décrit pas une instance**
⇒ sinon, on a fait une faute de conception

- le compteur d'instances est commun pour toutes les instances
⇒ static
- l'identifiant est spécifique à chaque instance
⇒ non static

FONCTION STATIC

- Boite à outils :
 - Génération de nom aléatoire (lettre aléatoire ou alternance voyelles/consonnes)
 - Distance entre Points (formulation alternative à celle intra-classe),
 - possibilité de définitions multiples pour prendre en compte des contraintes
 - optimisation ultérieure
- L'exemple de la classe Math

SINGLETON

- Pas d'accès au constructeur
- Méthode pour récupérer **LA SEULE** instance existante

```
1 public class Singleton {  
2     private static final Singleton INSTANCE = new Singleton();  
3  
4     private Singleton() {}  
5  
6     public static Singleton getInstance() {return INSTANCE;}  
7 }
```

Cas d'usage : définition d'un nouveau type

- Classe MonBooleen, constructeur privé, deux attributs static
- Accesseur static MonBooleen MonBooleen.isTrue(),
MonBooleen MonBooleen.getFalse()
- Dans le main, possibilité d'utiliser ==

UN FONCTIONNEMENT ASYMÉTRIQUE

- o les instances voient ce qui est static
- o les parties static ne voient pas les instances

```
1 public class Point{  
2     private static int cpt = 0;  
3     private int id;  
4     private double x,y;  
5     ...  
6     // Cas 1: OK méthode static , accès variable static  
7     public static int getCpt(){return cpt;}  
8     // Cas 2: OK méthode d'instance , accès variable static  
9     public int getCptInst(){return cpt;}}
```

UN FONCTIONNEMENT ASYMÉTRIQUE

- o les instances voient ce qui est static
- o les parties static ne voient pas les instances

```
1 public class Point{  
2     private static int cpt = 0;  
3     private int id;  
4     private double x,y;  
5     ...  
6     // Cas 1: OK méthode static , accès variable static  
7     public static int getCpt(){return cpt;}  
8     // Cas 2: OK méthode d'instance , accès variable static  
9     public int getCptInst(){return cpt;}  
10    // Cas 3 : KO méthode static , accès variable d'instance  
11    public static void getID(){return id;} // non sens!!
```

UN FONCTIONNEMENT ASYMÉTRIQUE

- o les instances voient ce qui est static
- o les parties static ne voient pas les instances

```
1 public class Point{  
2     private static int cpt = 0;  
3     private int id;  
4     private double x,y;  
5     ...  
6     // Cas 1: OK méthode static , accès variable static  
7     public static int getCpt(){return cpt;}  
8     // Cas 2: OK méthode d'instance , accès variable static  
9     public int getCptInst(){return cpt;}  
10    // Cas 3 : KO méthode static , accès variable d'instance  
11    public static void getID(){return id;} // non sens!!
```

Depuis le main :

```
1 Point p1 = new Point();  
2 // syntaxe naturelle :  
3 Point.getCpt();  
4 // syntaxe possible (mais pas recommandée)  
5 p1.getCpt();  
6 // syntaxe impossible (évidemment) :  
7 Point.getCptInst();
```

Quand on vous parle de *static*, n'oubliez pas :

- Ce sont des cas très particuliers
- Assez rare
- N'oubliez pas les bonnes pratiques de la POO!!!!!

2i002 Fiabilité & Exceptions

Vincent Guigue - vincent.guigue@lip6.fr



Le cours est inspiré de sources diverses: L. Denoyer, F Peschanski, E. Chailloux...

ENJEUX DU COURS

Améliorer la fiabilité des programmes développés

Etudions les outils qui vont nous donner confiance dans le code développé...

- ① Objectif 1 : code compilé = code fonctionnel
 - Les **erreurs de compilation sont les plus faciles à corriger...**
 - Respectons les règles de développement
 - Utilisons des outils pour **provoquer des erreurs de compilation** si le code prend mauvaise tournure...
 - Annotations (aide au compilateur)
- ② Objectif 2 : en cours d'exécution, détectons les erreurs et informons l'utilisateur
 - Fonctionnement des ruptures (exceptions)
 - Déclenchement, traitement....
- ③ Vers la programmation par contrat

FIABILITÉ (CORRECTION ET ROBUSTESSE)

[E. CHAILLOUX]

Spécification et réalisation :

- Un développement informatique est la réalisation effective d'une spécification.
- Un programme n'a pas d'erreur s'il est conforme à la spécification.
- Cette conformité peut être vérifiée par :
 - une preuve réalisée formellement (automatiquement ou à la main)
 - ou bien testée dans les deux cas en vérifiant les propriétés désirées.

FIABILITÉ = RESPECT DES RÈGLES DE DÉVELOPPEMENT

Idée

Pour éviter les erreurs, respecter les règles :

- Choix des noms pour comprendre qui fait quoi
- Classes & méthodes de taille raisonnable, limiter les accès public
 - Les opérations complexes sont déléguées à d'autres classes
 - Le client voit peu de choses : facile à comprendre, évite les failles
 - Taille limitée = on peut envisager de relire le code de la classe si nécessaire
- Evolutivité/architecture réfléchie (pour éviter les modifications ultérieures...), usage de final...
- Plus le code est clair, plus les erreurs sont faciles à voir

1 Voiture = 2 visions

- Vision client : accès aux contrôles comme un pilote (pédales, volant)
- Vision développeur : physique complexe à gérer pour obtenir un comportement réaliste

① Limiter la vision public :

- commandes pédales/volant ⇒ **public**
- calcul de la mise à jour de la position/direction, dérapage éventuel ⇒ **private**

② Limiter la taille des méthodes/classes

- Gestion de la physique = moteur physique ou géométrie dans l'espace... ⇒ **classes outils, vecteur, fichier** gérées à part

EXEMPLE SUR LA VOITURE 2/2

Idée

- ① Code morcelé = code plus lisible
- ② Code morcelé = code testable

Développer plusieurs main pour tester le bon fonctionnement des différentes fonctions basiques

En séparant la gestion de la physique de la voiture, il devient possible de tester chaque fonction du moteur physique...

- Plus facile de tester/corriger des méthodes basiques plutôt que l'ensemble d'une méthode très complexe

NB : il s'agit des prémisses de la programmation par contrat (cf plus loin & 3i002)

VÉRIFICATIONS STATIQUES

Idée

Faire en sorte de vérifier un maximum de chose au niveau de la compilation... Plus facile à corriger

- Par défaut le **compilateur** vérifie
 - **syntaxe** (;, parenthèses, accolades...)
 - **type** des variables et compatibilité avec les instances et méthodes appelées
 - **niveau d'accès** (aux méthodes, variables...)
- d'autres propriétés sont plus difficiles à montrer et nécessitent plus d'informations transmises au compilateur
 - **langage d'annotations**

ANNOTATIONS STANDARDS

- @Override : pour indiquer une redéfinition de méthode
- @Deprecated : pour indiquer un élément déprécié, engendre un warning
- @SuppressWarnings : pour désactiver ponctuellement des warnings

Certaines erreurs ne pose pas de problème de compilation mais provoque des comportements étranges lors de l'exécution... Ce sont les plus chères à corriger !

```
1  @Override  
2  public String toString() { // => erreur de compilation  
3      return "Point[" + x + ", " + y + "]";  
4  }
```

Erreur du type :

```
1 Point.java:23: method does not override or implement a method from  
2 @Override  
3 ^  
4 1 error
```

LANGAGES D'ANNOTATIONS POUR JAVA

[E. CHAILLOUX]

- Annotations en Java 1.5 : petit langage d'annotations intégré au langage depuis la 1.5
- JML (Java Modeling Language) est un langage de spécifications

```
1  /* ensures \result >= x && \result >= y &&
2           \forall integer z;
3           z >= x && z >= y ==> z >= \result;
4  */
5  public static int max(int x, int y) {
6      if (x>y) return x; else return x;
7  }
```

- permet d'être ensuite utilisées par des outils de vérification comme
 - ESC/Java 2 (Extended Static Checker for Java 2)
 - Krakatoa (Paris-sud 11, Inria)

NB : hors programme pour 2i002...

VÉRIFICATIONS DYNAMIQUES

Idées

Une fois la compilation passée, il reste de nombreuses erreurs possibles...

Les tests ne sont pas finis !

- ⇒ faire des tests sur les **arguments** des méthodes pour vérifier la faisabilité
- ⇒ faire des tests sur les **sorties** des méthodes pour vérifier la crédibilité des résultats obtenus

Exemples :

- Tester si la case demandée dans un tableau existe avant de retourner le résultat
- Tester si la valeur de retour de la fonction carre est bien positive
- Tester si l'argument de la division est bien différent de 0
- ...

RÉACTION EN CAS D'ÉCHEC DU TEST

Que faire ?

- utiliser une valeur spéciale : null, NaN

```
1 // dans un main
2     double r;
3     r = Math.asin(2.0);
4     // NB: asin n'accepte que des valeurs entre -1 et 1
5     System.out.println(r); // NaN
```

- effectuer une **rupture de calcul**

```
1 ArrayList<Double> arr = new ArrayList<Double>();
2 arr.add(1.); arr.add(2.); arr.add(4.);
3 System.out.println(arr.get(3));
4
5 //      Exception in thread "main" java.lang.IndexOutOfBoundsException
6 //                                         Index: 3, Size: 3
7 //      at java.util.ArrayList.RangeCheck(ArrayList.java:547)
8 //      at java.util.ArrayList.get(ArrayList.java:322)
9 //      at cours4.Test.main(Test.java:19)
```

EXCEPTIONS

Une **exception** est une rupture de calcul.

Elle est utilisée :

- pour éviter les erreurs de calcul
 - division par zéro
 - accès à la référence null
 - ouverture d'un fichier inexistant
 - ...
- comme style de programmation

En Java une exception est un objet

EXCEPTIONS : CRÉATION ET DÉCLENCHEMENT

Hiérarchie de classes (détails dans le cours suivant) :

```
1 java.lang.Object  
2     extended by java.lang.Throwable  
3         extended by java.lang.Exception  
4             extended by java.lang.RuntimeException
```

Création :

```
1 RuntimeException e = new RuntimeException("Division par zéro");
```

Déclenchement (throw) :

```
1 throw e;
```

Note : création & Déclenchement sont souvent combinés

```
1 throw new RuntimeException("Division par zéro");
```

EXEMPLE : GESTION D'UNE PILE D'ENTIERS

Rappel :

Une pile est une structure FILO : First In Last Out.

Le premier objet mis dans la pile sort en dernier.

```
1 public class Pile {  
2     private int[] items;  
3     private int niveau;  
4     public Pile(){items = new int[10]; niveau = 0;}  
5  
6     public void empiler(int item){  
7         items[niveau++] = item; // syntaxe compacte  
8     }  
9  
10    public int depiler(){  
11        return items[--niveau];  
12    }  
13 }
```

PILE : FONCTIONNEMENT

Usage normal :

```
1 public static void main(String [] args) {  
2     Pile p = new Pile();  
3     for(int i=0; i<6; i++)  
4         p.empiler(i);  
5  
6     for(int i=0; i<6; i++)  
7         System.out.println(p.depiler());  
8     // affichage de: 5 4 3 ... 0  
9 }
```

Erreur d'utilisation :

```
1 for(int i=0; i<6; i++)  
2     p.empiler(i);  
3 for(int i=0; i<7; i++)  
4     System.out.println(p.depiler());  
5 // Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException  
6 //      at cours4.Pile.depiler(Pile.java:13)  
7 //      at cours4.TestPile.main(TestPile.java:14)
```

```
8         // affichage de: 5 4 3 ... 0  
9 }
```

Erreur d'utilisation :

```
1 for(int i=0; i<6; i++)  
2     p.empiler(i);  
3 for(int i=0; i<7; i++)  
4     System.out.println(p.depiler());  
5 //  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException  
6 //      at cours4.Pile.depiler(Pile.java:13)  
7 //      at cours4.TestPile.main(TestPile.java:14)
```

La méthode dépiler a provoqué une rupture de calcul...

- Le programme s'est arrêté : les instructions suivantes ne sont pas exécutées
 - Bonne chose !! Si le programme continue avec une exécution faussée, l'erreur est plus dure à déceler
- Le message n'est pas clair

⇒ il faut détecter l'erreur et envoyer notre message : message clair + interruption garantie

PILE SÉCURISÉE

```
1 public void empiler(int item){  
2     if(niveau>=capacite)  
3         throw new RuntimeException("La pile est pleine:  
4             impossible d'ajouter des éléments");  
5     items[niveau++] = item;  
6 }  
7  
8 public int depiler(){  
9     if(niveau<=0)  
10        throw new RuntimeException("La pile est vide:  
11            impossible d'extraire des éléments");  
12    return items[--niveau];  
13 }
```

En cas de problème à l'exécution :

```
1 //Exception in thread "main" java.lang.RuntimeException:  
2 //      La pile est vide: impossible d'extraire des éléments  
3 //          at cours4.Pile.depiler(Pile.java:21)  
4 //          at cours4.TestPile.main(TestPile.java:14)
```

EXCEPTIONS : RÉCUPÉRATION D'EXCEPTIONS

+ structure try ... catch

```
1  try {
2      instruction
3  }
4  catch (<type exception> <variable>) {
5      instruction
6  }
7  [...]
8  finally {
9      instruction
10 }
```

La clause **finally** sera toujours exécutée avant de sortir du bloc try.

EXEMPLE EN JAVA

```
1 Pile p = new Pile();
2 for(int i=0; i<6; i++)
3     p.empiler(i);
4
5 try{
6     for(int i=0; i<8; i++){
7         System.out.println(p.depiler()); // rupture pour i=7
8     }
9
10    // cette instruction ne sera pas exécutée
11    System.out.println("Est-ce que je passe ici ?");
12 }catch(Exception e){ // récupération de la rupture
13     System.out.println("Impossible de dépiler => Exception");
14     System.out.println("Intercepté une erreur et poursuite du programme");
15 }
16
17 // cette instruction sera exécutée
18 System.out.println("Je suis passé ici");
```

INSTRUCTION finally

Définition

Dans ce bloc figure du code qui sera exécuté de toute façon ;

```
1 Pile p = new Pile();
2 for(int i=0; i<6; i++)
3     p.empiler(i);
4
5 try{
6     // on essaie limite = 4 et limite = 8
7     for(int i=0; i<limite; i++){
8         System.out.println(p.depiler());
9     }
10 }catch(Exception e){
11     System.out.println("Impossible de dépiler => Exception");
12     System.out.println("intercept de l'erreur et poursuite du pg");
13 }finally{
14     System.out.println("toto"); // toujours exécuté
15 }
```

- Usage peu courant : principalement pour la fermeture de fichier/socket

ATTENTION AUX PORTÉES DES VARIABLES

Les blocs limitent la portée des variables

Dans le catch, nous n'avons pas accès aux variables déclarées dans le try

```
1 try{  
2     int i = 7;  
3     ...  
4 }catch(Exception e){  
5     System.out.println(i); // ne compile pas: i n'existe pas!  
6 }  
// Bonne solution:  
7 int i = 0; // déclaration avant  
8 try{  
9     i = 7; // initialisation ici  
10    ...  
11 }catch(Exception e){  
12     System.out.println(i); // OK  
13 }  
14 }
```

Souvent une source de problèmes avec les fichiers (pour les fermer en cas d'interruption). cf prochain cours.

2i002 - Exercices d'application

Vincent Guigue



SÉLECTION DE MÉTHODES

```
1 public class Truc{  
2     public Truc(){ }  
3     public maMethode(int i){  
4         System.out.println("je passe dans : maMethode(int i)");  
5     }  
6     public maMethode(double d){  
7         System.out.println("je passe dans : maMethode(double d)");  
8     }  
9     public maMethode(double d1, double d2){  
10        System.out.println("je passe dans : maMethode(double d1, double d2)");  
11    }  
12    public maMethode(int i1, int i2, int i3){  
13        System.out.println("je passe dans : maMethode(int i1, int i2, int i3)");  
14    }  
15 }
```

```
1     Truc t = new truc();  
2     Truc t2 = new Truc(2);  
3     double deux = 2;  
4     int i = 2.5;  
5     t.maMethode(2);  
6     t.maMethode(deux);  
7     t.maMethode(2.);  
8     t.maMethode(1, 2);  
9     t.maMethode(1, 2, 3);  
10    t.maMethode(1., 2, 3);
```

REFLEXIONS *static*

On considère la classe **Chien**. Pour chacune des expressions suivantes, dire si ce sont des méthodes ou des attributs... Liés à l'instance (I) ou à la classe (C) :

nom	chercherLivreSurChiens()
aboyer()	vidéothèqueSurChiens
siteWebSurChiens	metsPrefere
siteWebDuChien	bibliographieSurChiens
siteWebSPA	dateNaissance
ageMax	age()
couleurDuPoil	manger()
courir()	regarderDVD()

EST CE QUE ÇA COMPILE ?(1)

But : créer des personnes avec des noms tirés aléatoirement

```
1 public class Personne {  
2     private String nom;  
3  
4     public Personne() {  
5         this("Individu");  
6         nom = nom +  
7             tirage() + tirage() + tirage();  
8     }  
9     public Personne(String nom) {  
10        this.nom=nom;  
11    }  
12    private char tirageLettre(){  
13        return (char) ((int) (Math.random()*26) + 'A');  
14    }  
15 }
```

EST CE QUE ÇA COMPILE ?(2)

But : créer des personnes avec des noms tirés aléatoirement

```
1 public class Personne {  
2     private String nom;  
3  
4     public Personne() {  
5         this("Individu" +tirage()+tirage()+tirage());  
6     }  
7     public Personne(String nom) {  
8         this.nom=nom;  
9     }  
10    private char tirageLettre(){  
11        return (char) ((int) (Math.random()*26) + 'A');  
12    }  
13 }
```

EST CE QUE ÇA COMPILE ?(3)

But : créer des personnes avec des noms tirés aléatoirement

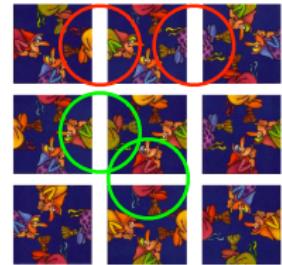
```
1 public class Personne {  
2     private String nom;  
3  
4     public Personne() {  
5         this("Individu" +tirage()+tirage()+tirage());  
6     }  
7     public Personne(String nom) {  
8         this.nom=nom;  
9     }  
10    private static char tirageLettre(){  
11        return (char) ((int) (Math.random()*26) + 'A');  
12    }  
13 }
```

SYNTAXE DES MATRICES ET PROGRAMME MYSTÈRE

```
1 int s = 3;
2 int [][] mat = new int [s][s];
3 for(int i=0; i<s; i++)
4     for(int j=0; j<s; j++)
5         mat[i][j] = i*s+j;
6
7 int [][] mat2 = {{1,2,3},{4,5,6},{7,8,9}};
8 for(int i=0; i<s; i++){
9     for(int j=0; j<s; j++)
10        System.out.print(mat2[i][j]+"\u00a0");
11    System.out.println();
12 }
13
14 int [][] mat3 = {{1,2},{3,4,5,6},{7,8,9}};
15 for(int i=0; i<s; i++){
16     for(int j=0; j<s; j++)
17         System.out.print(mat3[i][j]+"\u00a0");
18     System.out.println();
19 }
20
21 int [][] mat4 = new int [5][];
22 System.out.println(mat4);
23 System.out.println(mat4[0]);
24 System.out.println(mat4[0][0]);
```

EXEMPLE : RÉSOLUTION D'UN CASSE TÊTE (ARCHITECTURE & SYNTAXE)

Hexen Spiele :



But : mettre les cartes dans une disposition telle que toutes les sorcières soient entières et bien formées (tête + derrière de la même couleur)

Proposition d'algorithme :

- ① Disposer aléatoirement les 9 cartes
- ② Appliquer une rotation sur chaque carte pour minimiser le nombre d'incompatibilité
- ③ Compter les erreurs
 - S'il reste des erreurs, recommencer

Eléments et fonctionnalités associées

- Pièce : *rotation*
- Jeu (composé de Pièce) : *comptage des erreurs, disposition aléatoire, optimisation de la rotation d'une pièce*

Justification

- Distinction entre les différents éléments physiques

HEXEN SPIELE, MODÉLISATION

Eléments et fonctionnalités associées

- **Pièce** : *rotation*
distinguer la **Pièce** du **Jeu** permet de simplifier le code dans chaque classe
- **Jeu** (composé de **Pièce**) : *comptage des erreurs, disposition aléatoire, optimisation de la rotation d'une pièce*

Justification

- Distinction entre les différents éléments physiques

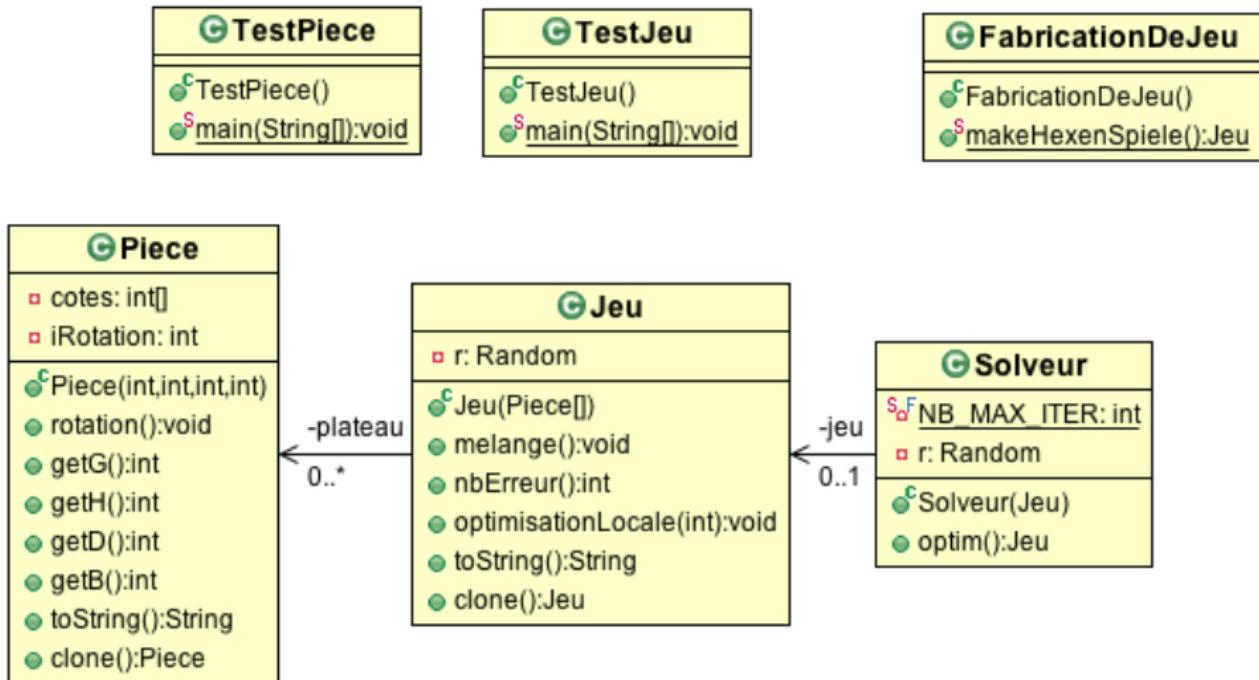
Eléments et fonctionnalités associées

- **Pièce** : *rotation*
- **Jeu** (composé de **Pièce**) : *comptage des erreurs, disposition aléatoire, optimisation de la rotation d'une pièce*
- **Solveur** : *optimisation*

Justification

- Distinction entre les différents éléments physiques
- Construction à part du solveur :
 - Possibilité d'en développer d'autres plus tard
 - Possibilité d'utiliser le Jeu sans solveur

ARCHITECTURE FINALE



POINT *journalisé*

Sécurisation de la classe Point par journalisation (sauvegarde de toutes les valeurs successives du point)

```
1 public class Point{  
2     private double x,y;  
3  
4     public Point(double x, double y) {  
5         this.x = x;  
6         this.y = y;  
7     }  
8  
9     public double getX() { return x; }  
10    public double getY() { return y; }  
11  
12    public String toString() {  
13        return "Point[x=" + x + ",y=" + y + "]";  
14    }  
15  
16    public void move(double dx, double dy){  
17        x+=dx; y+=dy;  
18    }
```

Les questions qui se posent :

- Sous quelle forme sauvegarder les différentes positions du Point ?
- Où faire la sauvegarde ?
- Implémentation :

POINT *journalisé*

Les questions qui se posent :

- Sous quelle forme sauvegarder les différentes positions du Point ?

Plusieurs solutions mais le plus simple... `ArrayList<Point>`

- Gestion automatique de la taille de la sauvegarde
- Affichage facile, ce sont des points

- Où faire la sauvegarde ?
- Implémentation :

POINT *journalisé*

Les questions qui se posent :

- Sous quelle forme sauvegarder les différentes positions du Point ?
- Où faire la sauvegarde ?
Dans les modifieurs. Il restera toujours une faille : à l'intérieure de la classe, il faut se forcer à les utiliser.
- Implémentation :

POINT *journalisé*

Les questions qui se posent :

- Sous quelle forme sauvegarder les différentes positions du Point ?
- Où faire la sauvegarde ?
- Implémentation :
Dans une méthode privée qui pourra être appelée dans les accesseurs

POINT journalisé : RÉSULTAT

```
1 public class Point{  
2     private double x,y;  
3     private ArrayList<Point> sauvegarde;  
4  
5     public Point(double x, double y) {  
6         this.x = x; this.y = y;  
7         sauvegarde = new ArrayList<Point>(); // ne pas oublier!  
8     }  
9  
10    private void save(){ sauvegarde.add(this.clone());}  
11  
12    public void setX(double x){save(); this.x = x;}  
13    public void setY(double y){save(); this.y = y;}  
14  
15    public Point clone(){return new Point(x,y);}  
16    // accesseurs , toString ...  
17  
18    public void move(double dx, double dy){  
19        x+=dx;y+=dy; // ATTENTION !  
20        setX(x+dx); setY(y+dy);  
21    }
```