2I002
FIABILITÉ & EXCEPTIONS

Vincent Guigue & Christophe Marsala

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

SCIENCES SORBONNE UNIVERSITÉ
LIP6
MLIA Machine Learning & Deep Learning for Information Access

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

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

`final` ⇒ attributs static = CONSTANTES

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).
Exemple : `Math.PI`, sécurisation = impossibilité de modifier

Note : une constante est indépendante des instances ⇒ `static`
Usage : une constante est définie en majuscule

`final` ⇒ attributs static = CONSTANTES

Pour sécuriser le code, interdisons les modifications de certaines valeurs (notamment les constantes).
Exemple : `Math.PI`, sécurisation = impossibilité de modifier

Note : une constante est indépendante des instances ⇒ `static`
Usage : une constante est définie en majuscule

```
1  public class MaClasse{
2    public final static int MACONSTANTE = 10;
3    ...
```

Usage:

- Constantes *universelles* (`Color.RED`, `Color.YELLOW`, `Math.PI`, `Double.POSITIVE_INFINITY`...)
- Typologie (type de codage d'un pixel, organisation du BorderLayout)...
- Bornes algorithmiques (`NB_ITER_MAX`, `TAILLE_MAX`...)

Idée : protéger ses objets... Et ses programmes

Initialiser les valeurs des attributs sans pouvoir les modifier ensuite
Exemple : `String`

```
1  public class Point{
2    public final double x,y;
3    public Point(double x, double y){
4      this.x = x; this.y = y;
5    }
6    // interdiction de modifier x, y dans la suite
7    // (et chez le client)
8  }
```

- Interdiction de modifier `x` et `y` dans les méthodes
  (pas de setter, pas de translation...)
- Modification d'un `Point` = création d'une nouvelle instance
- Possibilité de laisser les attributs `public`... Puisque non modifiable
- Sécurité lorsqu'un objet est passé en argument de méthode

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

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 prémices de la programmation par contrat (cf plus loin & 3i002)

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)
  - testée dynamiquement durant l'exécution en vérifiant les propriétés désirées.

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

- `@Override` : pour indiquer une redéfinition de méthode
- `@Deprecated` : pour indiquer un élément déppprécié, 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 toStrig() { // => erreur de compilation
3         return "Point [x=" + x + ", y=" + y + "]";
4     }
```

Erreur du type:

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

- 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...

Une fois la compilation passée, il reste de nombreuses erreurs possibles...
Les tests ne sont pas finis!

$\Rightarrow$ faire des tests sur les **arguments** des méthodes pour vérifier la faisabilité

$\Rightarrow$ 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
- ...

Que faire?

- utiliser une valeur spéciale : `null`, `NaN`

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

- effectuer une rupture de calcul

```
   ArrayList<Double> arr = new ArrayList<Double>();
   arr.add(1.);arr.add(2.);arr.add(4.);
   System.out.println(arr.get(3));

//Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 3, Size
//       at java.util.ArrayList.RangeCheck(ArrayList.java:547)
//       at java.util.ArrayList.get(ArrayList.java:322)
//       at cours4.Test.main(Test.java:19)
```

⇒ Vous avez observé ces réponses sur d'autres objets...

il reste à les provoquer vous-même.

Exploitation du `null`:

```
1 // client
2 Point p = Point.rechercheePositif(tab);
3 // soit p pointe vers une instance
4 // positive... soit p est null
5 if(p == null){
6   System.out.println
7           ("Pas de point positif");
8 }
```

Exploitation du `null`:

```
// client
Point p = Point.rechercheDePositif(tab);
// soit p pointe vers une instance
// positive... soit p est null
if(p == null){
  System.out.println
          ("Pas de point positif");
}
```

```
// class Point
public static Point
        rechercheDePositif(Point[] t){
  for(Point p:t)
    if(p.x > 0 && p.y > 0)
      return p;
  return null;
}
```

Exploitation du `null`:

```
1 // client
2 Point p = Point.rechercheDePositif(tab);
3 // soit p pointe vers une instance
4 // positive... soit p est null
5 if(p == null){
6   System.out.println
7           ("Pas de point positif");
8 }
```

```
1 // class Point
2 public static Point
3         rechercheDePositif(Point[] t){
4   for(Point p:t)
5     if(p.x > 0 && p.y > 0)
6       return p;
7   return null;
8 }
```

Exploitation du `Nan`:

```
1 // client
2 double d = TestNan.monCalcul(2);
3
4 if(Double.isNaN(d))
5   System.out.println
6         ("Calcul impossible");
7 else
8   System.out.println(d);
```

Exploitation du `null`:

```
// client
Point p = Point.rechercheDePositif(tab);
// soit p pointe vers une instance
// positive... soit p est null
if(p == null){
  System.out.println
          ("Pas de point positif");
}
```

```
// class Point
public static Point
        rechercheDePositif(Point[] t){
  for(Point p:t)
    if(p.x > 0 && p.y > 0)
      return p;
  return null;
}
```

Exploitation du `Nan`:

```
// client
double d = TestNan.monCalcul(2);

if(Double.isNaN(d))
  System.out.println
        ("Calcul impossible");
else
  System.out.println(d);
```

```
public static double
          monCalcul(double a){
  if(a>0)
    return 2*Math.sqrt(a) -5;
  return Double.NaN;
}
```

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

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

- Création de l'objet *plantage*
- Déclenchement d'un plantage...
- ... Et récupération de ce plantage!

Hiérarchie de classes (détails dans le cours suivant):

```
java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.lang.RuntimeException
```

Création d'une exception (= utilisation d'un objet existant) :

```
RuntimeException e = new RuntimeException("Division par zero");
```

Déclenchement (`throw`) :

```
throw e;
```

Note: création & Déclenchement sont souvent combinés

```
throw new RuntimeException("Division par zero");
```

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 }
```

Usage normal :

```
1  public static void main(String[] args) {
2        Pile p = new Pile();
3        for(int i=0; i<6; i++) // mettre 6 entiers
4            p.empiler(i);
5
6        for(int i=0; i<6; i++) // enlever 6 entiers
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++) // mettre 6 entiers
2    p.empiler(i);
3  for(int i=0; i<7; i++) // enlever 7 entiers
4    System.out.println(p.depiler());
5  //  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
6  //       at cours4.Pile.depiler(Pile.java:13)
7  //       at cours4.TestPile.main(TestPile.java:14)
```

Erreur d'utilisation:

```
1 for(int i=0; i<6; i++) // mettre 6 entiers
2   p.empiler(i);
3 for(int i=0; i<7; i++) // enlever 7 entiers
4   System.out.println(p.depiler());
5 //  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
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

$\Rightarrow$ il faut: (1) détecter l'erreur et (2) envoyer un message clair

```java
public void empiler(int item){
  if(niveau>=capacite)
    throw new RuntimeException("La pile est pleine:
  impossible d'ajouter des elements");
  items[niveau++] = item;
}

public int depiler(){
  if(niveau<=0)
    throw new RuntimeException("La pile est vide:
  impossible d'extraire des elements");
  return items[--niveau];
}
```

En cas de problème à l'exécution :

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

+ structure `try ... catch`

```
try {
        ...instructions...
}
catch (<type exception> <variable>) {
        ...instructions...
}
finally {
        ...instructions...
}
```

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 executee
11    System.out.println("Est ce que je passe ici?");
12 }catch(RuntimeException e){ // récupération de la rupture
13     System.out.println("Impossible de depiler => Exception");
14     System.out.println("Intercep. de l'erreur et poursuite du pg");
15 }
16
17 // cette instruction sera executee
18 System.out.println("Je suis passe ici");
```

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(RuntimeException e){
11     System.out.println("Impossible de depiler => Exception");
12     System.out.println("intercept. de l'erreur et poursuite du pg");
13 }finally{
14     System.out.println("toto"); // toujours execute
15 }
```

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

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(RuntimeException e){
5   System.out.println(i); //
```

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

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