SHAKE THE FUTURE

CENTRALE NANTES

Programmation Orientée Objet - Chapitre 5 – Exceptions, Rappels, Conclusion

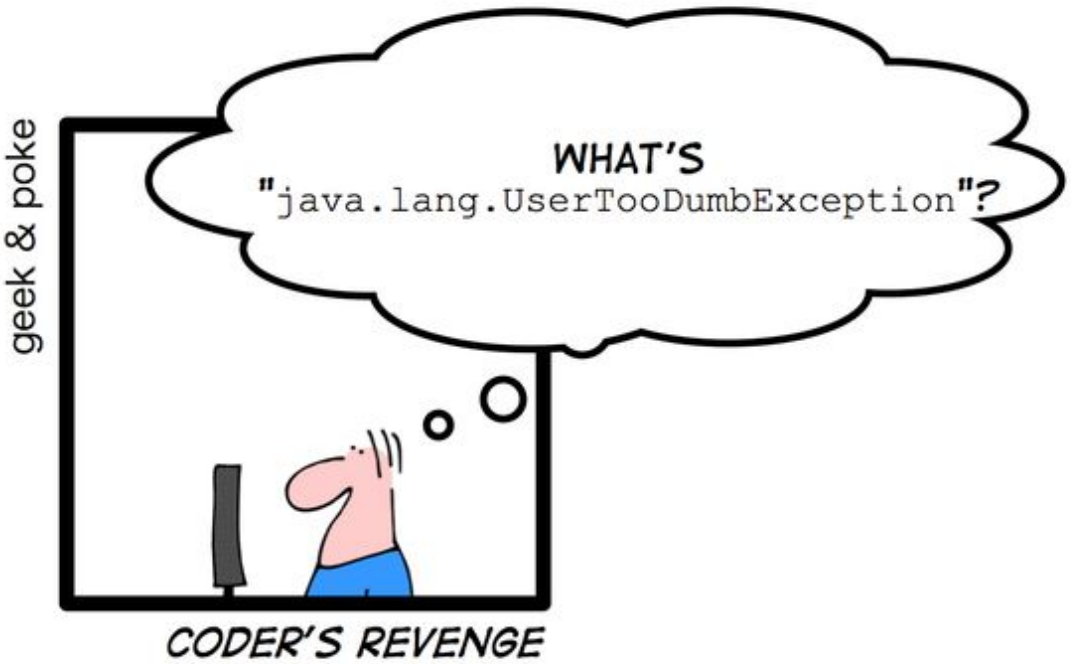
Jean-Marie Normand
Bâtiment E - Bureau 211
jean-marie.normand@ec-nantes.fr

CENTRALE NANTES

Plan du cours I

Plan

Introduction



Les exceptions

- mécanisme très fréquemment utilisé en java
- les exceptions se rencontrent (malheureusement ?) dans de nombreuses situations

```
class UncatchedException {
  public static void main(String[] args) {
    int[] tab = new int[18];
    // on fait volontairement une betise
    System.out.println(tab[18]);
  }
}
```

Les exceptions

- mécanisme très fréquemment utilisé en java
- les exceptions se rencontrent (malheureusement ?) dans de nombreuses situations

```
class UncatchedException {
  public static void main(String[] args) {
    int[] tab = new int[18];
    // on fait volontairement une betise
    System.out.println(tab[18]);
  }
}
```

Exécution

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 18
        at UncatchedException.main(UncatchedException.java:6)
```

Exemple introductif

Fonction qui calcule l'inverse d'un nombre réel

```
public static double inverse(double x) {
    // on suppose que le premier argument du programme est un entier
    if(x == 0) {
      // Erreur !!
    }
    else {
      return (1.0/x);
    }
}
```

Concrètement que peut-on faire en cas d'erreur ?

Exemple introductif

Fonction qui calcule l'inverse d'un nombre réel

```
public static double inverse(double x) {
    // on suppose que le premier argument du programme est un entier
    if(x == 0) {
      // Erreur !!
    }
    else {
      return (1.0/x);
    }
}
```

Concrètement que peut-on faire en cas d'erreur ?

1. retourner une valeur choisie à l'avance (par exemple `Double.MAX_VALUE`), **mais** cela :
   - **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
   - retourne de toute façon un **résultat incorrect/inexact**
   - suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

Exemple introductif II

Concrètement que peut-on faire en cas d'erreur ?

2. afficher un message d'erreur (`System.out.println("Erreur!");`) : OK mais que retourne t'on ? La méthode **doit** retourner une valeur !
   - de plus cela est très mauvais car cela produit des effets de bord
   - affichage dans le terminal alors que **ce n'est pas du tout le rôle** de `inverse`
   - la fonction `inverse` n'a peut-être pas toutes les cartes en main pour gérer le cas d'erreur

Exemple introductif II

Concrètement que peut-on faire en cas d'erreur ?

2. afficher un message d'erreur (`System.out.println("Erreur!");`) : OK mais que retourne t'on ? La méthode **doit** retourner une valeur !
   - de plus cela est très mauvais car cela produit des effets de bord
   - affichage dans le terminal alors que **ce n'est pas du tout le rôle** de `inverse`
   - la fonction `inverse` n'a peut-être pas toutes les cartes en main pour gérer le cas d'erreur
3. retourner un code d'erreur (déclarer `inverse` comme retournant un `boolean` plutôt qu'un `double`)

   if inverse(x,y)
   inverse(x,y)
   - cette solution est **meilleure** car elle laisse à la fonction qui appelle `inverse` le soin de décider quoi faire en cas d'erreur
   - mais c'est assez lourd à gérer (en cas d'appels en cascades à des fonctions)
   - écriture peu intuitive : `if(inverse(x,y))` au lieu de

Comment gérer les erreurs ?

On veut un mécanisme qui permette de :

- signaler une situation d'erreur sans produire d'effets de bord indésirés
- gérer une erreur à un autre endroit que là où elle est détectée
- garder une souplesse d'utilisation, une utilisation intuitive dans le programme

Solution : lever (déclencher) une exception

Ce mécanisme permet de prévoir une erreur à un endroit et de la gérer à un autre endroit dans le programme

Contrôle du comportement à l'exécution (exemple simplifié)

```
class ControleComportementExecution {
  public static void main(String[] args) {
    // on suppose que le premier argument du programme est un entier
    int valeur=0;
    try {
      valeur = Integer.parseInt(args[0]);
    }
rattraper l'erreur <-catch (NumberFormatException e) {
      System.out.println("le programme doit etre appele avec un argument de type entier");
      return;
    }
    // suite du programme
    System.out.println("vous avez donne la valeur : "+valeur);
  }
}
```

terminal:
javac CCE.java —> .class
java CCE 3(dans args) dans main

Contrôle du comportement à l'exécution (exemple simplifié)

```
class ControleComportementExecution {
  public static void main(String[] args) {
    // on suppose que le premier argument du programme est un entier
    int valeur=0;
    try {
      valeur = Integer.parseInt(args[0]);
    }
    catch (NumberFormatException e) {
      System.out.println("le programme doit etre appele avec un argument de type entier");
      return;
    }
    // suite du programme
    System.out.println("vous avez donne la valeur : "+valeur);
  }
}
```

Que se passe-t-il pour les exécutions suivantes ?

```
java ControleComportementExecution 43
java ControleComportementExecution 43k
```

Contrôle du comportement à l'exécution (exemple simplifié)

Que se passe-t-il pour les exécutions suivantes ?

```
java ControleComportementExecution 43
java ControleComportementExecution 43k
```

```
    Exception$ java ControleComportementExecution 43
vous avez donne la valeur : 43
    Exception$ java ControleComportementExecution 43k
le programme doit etre appele avec un argument de type entier
```

```
      return;
    }
    // suite du programme
    System.out.println("vous avez donne la valeur : "+valeur);
  }
}
```

Les exceptions : définition

- Les exceptions permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'exécution d'une portion de code qui perturbe l'exécution nominale d'un programme
- Une exception est un signal
  - qui indique que quelque chose d'exceptionnel (par exemple une erreur) s'est produit
  - qui interrompt le flot d'exécution normal du programme
- **lancer** (`throw`) une exception revient à signaler quelque chose
- **attraper** (`catch`) une exception permet d'exécuter les actions nécessaires au traitement de la situation particulière

Les exceptions : mécanisme

Le mécanisme de la gestion des exceptions :

- lorsqu'une situation exceptionnelle est rencontrée **une exception est lancée**
- si cette exception n'est pas attrapée dans le bloc de code dans lequel elle se trouve, elle est propagée au bloc de code supérieur et ainsi de suite...
- même principe pour les méthodes...
- si une exception n'est jamais attrapée :
  - propagation jusqu'à la méthode `main`
  - affichage d'un message d'erreur et de la pile d'appel
  - arrêt de l'exécution du programme

Les exceptions : mécanisme

Le mécanisme de la gestion des exceptions :

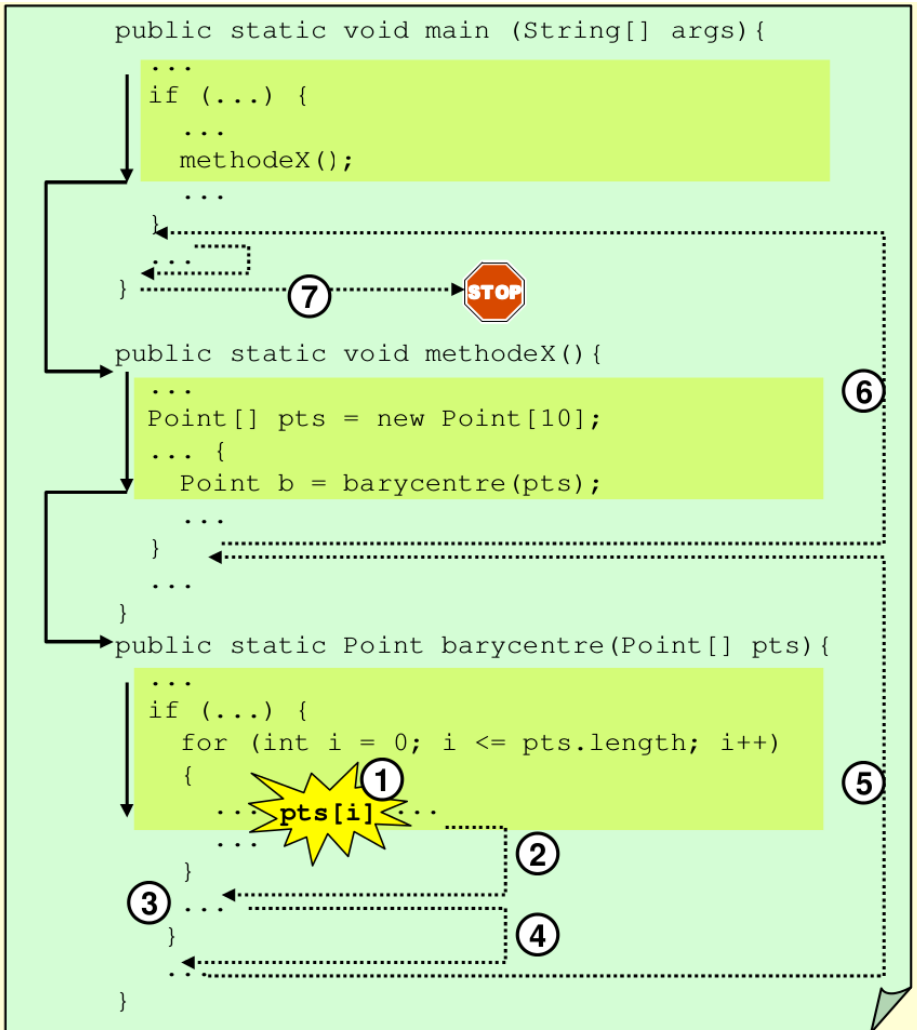
- lorsqu'une situation exceptionnelle est rencontrée **une exception est lancée**
- si cette exception n'est pas attrapée dans le bloc de code dans lequel elle se trouve, elle est propagée au bloc de code supérieur et ainsi de suite...
- même principe pour les méthodes...
- si une exception n'est jamais attrapée :
  - propagation jusqu'à la méthode `main`
  - affichage d'un message d'erreur et de la pile d'appel
  - arrêt de l'exécution du programme

Rappel slide 6

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 18
        at UncatchedException.main(UncatchedException.java:6)
```

CENTRALE NANTES

Les exceptions : mécanisme



1. Dépassement d'index, arrêt de l'exécution normale et *lancement* de `ArrayOutOfBoundsException`
2. Transmission du contrôle au bloc de niveau supérieur
3. Si il y a du code pour traiter (*attraper*) l'exception reprise du flot d 'exécution normal
4. Sinon on recommence comme en 2
5. Si aucun code dans la méthode pour traiter l'exception le contrôle est transmis au niveau de la méthode appelante et on recommence comme en 2
6. Et ainsi de suite...
7. Jusqu'à ce que l'on aboutisse au bloc du programme principal, alors arrêt de l'exécution et impression de la pile des appels.

Les objets `Exception`

- En Java, les exceptions sont des **objets**
- Toute exception est une instance d'une sous-classe de la classe `java.lang.Throwable`
- Celle-ci possède deux sous-classes standards :
  - `Error` : non récupérables, essentiellement des erreurs systèmes (mémoire, chargement dynamique)
  - `Exception` : récupérables ... ou pas
- Pour définir ses propres exceptions, on sous-classera `Exception` (pas forcément directement)

Les objets `Exception`

- puisque ce sont des objets, les exceptions ont des attributs et des méthodes (et on peut en ajouter à volonté dans les sous-classes)
- Dans `java.lang.Throwable` :
  - `message` : contient un message décrivant l'exception
  - `cause` : utilisé pour le chaînage des exceptions
  - `Throwable(msg)` : constructeur
  - `printStackTrace()` : affiche la pile d'appel
  - accesseurs, etc.
  - voir http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

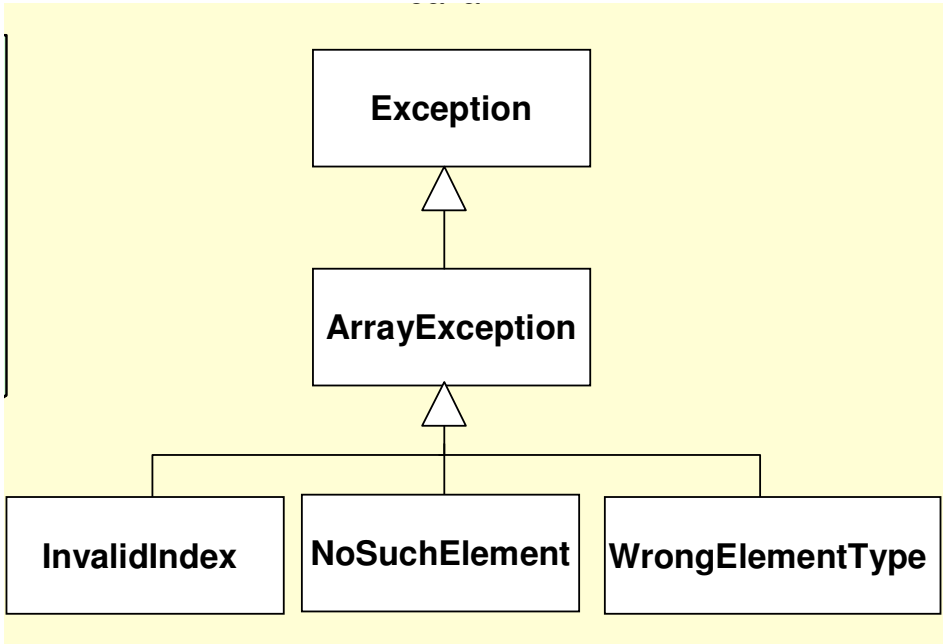
Pourquoi des objets Exception ?

- La représentation des exceptions sous forme d'objets permet de mieux structurer la description et le traitement des erreurs

"Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be instances of Throwable or anyThrowable descendant. As for other Java classes, you can create subclasses of the Throwable class and subclasses of yoursubclasses. `Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions.`"

tiré du *The Java Tutorial* de Mary Campione and Kathy Walrath

Structuration de la description des exceptions



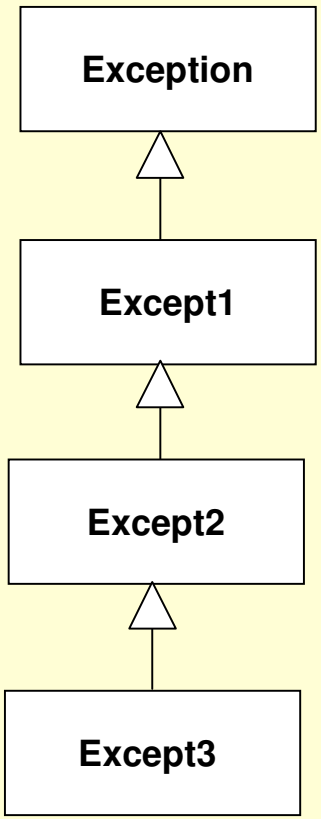
Spécifications d'erreurs plus précise

- on peut être amené à définir une hiérarchie de classes (qui n'ont pas forcément des attributs ou des méthodes différentes), uniquement dans un souci de structuration et de typage

Traitement des exceptions

- `try { ... }` définit un ensemble d'instructions susceptibles de déclencher des exceptions pour lesquelles une gestion est mise en œuvre
- cette gestion est réalisée par un ou plusieurs blocs `catch (TypeException e) { ... }` qui suivent le bloc `try`
- cela permet d'attraper les exceptions et d'exécuter du code spécifique
- un seul bloc `catch` est exécuté (le premier)
  - en fonction du type de l'exception
  - attention à l'ordre des `catch` !

De l'importance de l'ordre des blocs catch



Exception de type :**Except1**
ou **Except2** ou **Except3**

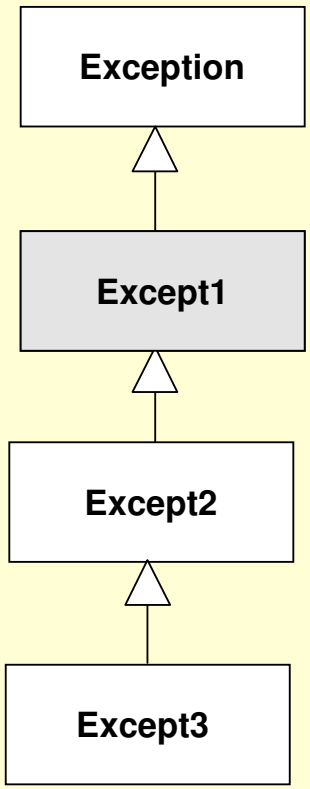
quel bloc `catch`
va la traiter ?

?

```
...
try {

    // séquence de code pouvant lever
    //des exceptions de type Except1,
    // Except2 ou Except3
    ...
}
catch (Except1 e1)
{
    // e1 est l'exception qui a été levée
    // dans le bloc try
    System.out.println(e1.getMessage());
    e1.printStackTrace();
    // ...
}
catch (Except2 e2)
{
  // ...
}
catch (Except3 e3)
{
  // ...
}
```

De l'importance de l'ordre des blocs catch



Exception de type :
Except1

type Except2

type Except1

```
...
try {

   // séquence de code pouvant lever
   //des exceptions de type Except1,
   // Except2 ou Except3
   ...
}
catch (Except2 e2)
{
   // e2 est l'exception qui a été levée
   // dans le bloc try
   System.out.println(e2.getMessage());
   e2.printStackTrace();
   // ...
}
catch (Except1 e1)
{
  // ...
}
catch (Except3 e3)
{
  // ...
}
```

De l'importance de l'ordre des blocs catch (1/2) I

```
class MyException1 extends Exception { }
class MyException2 extends MyException1 { }
class MyException3 extends MyException2 { }

public class MyExceptionTester {
  public void doIt(String ex) {
    // le code qui va lever les exceptions  try {
      if (ex.equals("Exception1")) {
        throw new MyException1();
      }
      if (ex.equals("Exception2")) {
        throw new MyException2();
      }
      if (ex.equals("Exception3")) {
        throw new MyException3();
      }
    }
```

De l'importance de l'ordre des blocs catch (1/2) II

```
    catch (MyException1 e) {
      System.out.println(e.getMessage());
      return;
    }
    catch (MyException2 e) {
      System.out.println(e.getMessage());
      return;
    }
    catch (MyException3 e) {
      System.out.println(e.getMessage());
      return;
    }
}
```

De l'importance des blocs catch (2/2)

```
public static void main(String[] args) {
  new MyExceptionTester().doIt(args[0]);
}
```

- ne compile même pas !

```
MyExceptionTester.java:29: exception MyException2 has already been caught
                catch (MyException2 e) {
                ^
MyExceptionTester.java:33: exception MyException3 has already been caught
                catch (MyException3 e) {
                ^
```

- l'ordre d'héritage doit être respecté : du plus spécialisé au plus général

Traitement des exceptions

Résumé :

- pas nécessaire d'avoir une clause `catch` pour chaque type possible d'exception
- si aucun bloc `catch` n'attrape l'exception, celle-ci est propagée vers la méthode appelante
- si une exception n'est attrapée nulle part, alors un message d'erreur est généré en affichant le nom de l'exception (avec ses paramètres) ainsi que la pile d'appels ; l'exécution du programme est interrompue

Traitement des exceptions : finalisation

- Les clauses `catch` sont suivies de manière optionnelle par un bloc `finally` qui contient du code à exécuter quelle que soit la manière dont le bloc `try` a été quitté
- Il permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive :
  - le bloc `try` s'exécute normalement sans levée d'exception
  - fin du bloc `try` (ou `return`, `break`...)
  - le bloc `try` lève une exception qu'il attrape
  - le bloc `try` lève une exception qu'il n'attrape pas

Traitement des exceptions : finalisation

- rassembler dans un seul bloc un ensemble d'instructions qui auraient pu être dupliquées
- effectuer des traitements après le bloc `try`

```
...
 try {
    // ouvrir un fichier
    // effectuer des traitements
    // susceptibles
    // de lever une exception
    // fermer le fichier
 }
 catch (CertaineException e)
 {
   // traiter l'exception
   // fermer le fichier
 }
 catch (AutreTypeException e)
 {
   // traiter l'exception
   // fermer le fichier
 }
```

```
...
 try {
    // ouvrir un fichier
    // effectuer des traitements
    //susceptibles
    // de lever une exception
 }
 catch (CertaineException e)
 {
   // traiter l'exception
 }
 catch (AutreTypeException e)
 {
   // traiter l'exception
 }
 finally {
   //fermer le fichier
 }
```

regrouper les codes

Exemple I

```
public class MyEException1 extends Exception { }
public class MyEException2 extends Exception { }

public class ExempleException {
  public static void main(String[] args) throws MyEException2 {
    try {
      new ExempleException().doIt(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException ex) {
      System.out.println("il fallait mettre un argument lors de
     l'appel");
    }
  }
  public void doIt(String e) throws MyEException2 {
    System.out.println("marqueur 1");
    try {
      if (e.equals("STOP")) {
        throw new MyEException1();
      }
```

Exemple II

```
      if (e.equals("2")) {
        throw new MyEException2();
      }
      System.out.println("marqueur 2");
    } // fin du bloc try
    catch (MyEException1 ex) {
      System.out.println("marqueur 3");
      return;
    }
    finally {
      System.out.println("marqueur finally");
    }
  }
}
```

Cas d'exécution

```
[mac-infmat07:OBJET/OBJET/code] % java ExempleException
il fallait mettre un argument lors de l'appel
[mac-infmat07:OBJET/OBJET/code] % java ExempleException  q
marqueur 1
marqueur 2
marqueur finally
[mac-infmat07:OBJET/OBJET/code] % java ExempleException STOP
marqueur 1
marqueur 3
marqueur finally
[mac-infmat07:OBJET/OBJET/code] % java ExempleException 2
marqueur 1
marqueur finally
Exception in thread "main" MyEException2
        at ExempleException.doIt(ExempleException.java:25)
        at ExempleException.main(ExempleException.java:11)
```

Lancement d'exceptions

- L'instruction `throw` *`unObjetException`* permet de lancer une exception
- *`unObjetException`* doit être une instance d'une sous-classe de `Throwable`
- Quand une exception est lancée :
  - l'exécution normale du programme est interrompue
  - la JVM cherche un bloc `catch` approprié en remontant la pile d'appels
  - tous les blocs `finally` rencontrés au cours de cette propagation sont exécutés

Déclaration des exceptions

- Toute méthode susceptible de lever une exception doit :
  - soit l'**attraper**
  - soit la **déclarer explicitement** c'est-à-dire comporter dans sa signature l'indication que l'exception peut être provoquée : clause `throws`
- Les exceptions déclarées dans la clause `throws` d'une méthode sont :
  - les **exceptions levées dans la méthode et non attrapées**
  - les **exceptions levées dans les méthodes appelées par la méthode et non attrapées par celle-ci**

Exemple

```
public class TestException {
  public void method1() throws MonException {
    throw new MonException();
  }

  public void method2() {
    method1();
   }
}
```

faux, pas de block «catch »

Erreur de compilation

```
TestException.java:10: unreported exception MonException;
must be caught or declared to be thrown
                method1();
                       ^
1 error
```

Exemple : methode2() : correction

Deux solutions :

1. attraper l'exception

```
public void method2() {
  try {
    method1();
  }
  catch (MonException e) {
    // ...
  }
}
```

2. la déclarer

```
public void method2() throws MonException {
  method1();
}
```

Déclaration des exceptions

- Plusieurs classes d'exceptions peuvent être indiquées dans la clause `throws` d'une déclaration de méthode
- La classe utilisée pour déclarer les exceptions peut être une superclasse de la classe de l'exception effectivement lancée
- À la limite, toutes les méthodes pourraient faire `throws Exception`

Intérêt de la déclaration des exceptions

- **Celui qui écrit une méthode** doit être conscient de toutes les exceptions levées par les méthodes qu'il appelle. Pour ces exceptions, il doit choisir entre les traiter (bloc `try/catch`) ou les déclarer (instruction `throws`). Il ne peut pas les ignorer !
- **Celui qui utilise la méthode** apprend grâce aux clauses `throws` quelles sont les exceptions susceptibles d'être levées par cette méthode et les méthodes appelées

Compromis sur la déclaration des exceptions

- Pour simplifier l'écriture des programmes (et permettre une future extensibilité), les exceptions "standard" n'ont pas besoin d'être déclarées :
  - exceptions définies comme sous-classes de `Error`
  - exceptions définies comme sous-classes de `RuntimeException` (par exemple `ArrayOutOfBoundsException`, `NullPointerException`...)
- En revanche, il reste interdit de faire ceci :

Blocs `catch` vides

- un bloc `catch` vide est extrêmement suspect : –> throw (MyException)
  - détruit la finalité des exceptions qui est d'obliger le programmeur à traiter les situations exceptionnelles qu'elles représentent
- 2 types de solutions
  - gérer l'erreur comme une exception simplement déclarée (`throws`)
  - justifier explicitement une raison de ne rien faire

Gestion des erreurs : sans exceptions

Séparation du code de gestion des erreurs du code « normal »

```
errorCodeType readFile {
     initialize errorCode = 0;
     open the file;
     if (theFileIsOpen) {
         determine the length of the file;
         if (gotTheFileLength) {
           allocate that much memory;
             if (gotEnoughMemory) {
                 read the file into memory;
                 if (readFailed) {
                     errorCode = -1;
                 }
             } else {
                 errorCode = -2;
             }
         } else {
             errorCode = -3;
         }
         close the file;
         if (theFileDidntClose && errorCode == 0) {
             errorCode = -4;
         } else {
             errorCode = errorCode and -4;
         }
     } else {
         errorCode = -5;
     }
     return errorCode;
 }
```

- ajouter dans le code de la méthode des instructions pour détecter, rapporter et gérer les erreurs, utilisation de code d 'erreur en retour des fonctions
- augmentation conséquente de la taille du code (de 7 lignes de codes on passe à 29 lignes, augmentation de plus de 400% !!)
- grande perte de lisibilité (le code devient un plat de spaghetti)
- que faire pour les fonctions qui doivent renvoyer un résultat ?

➔ **gestion des erreurs souvent négligée par les programmeurs**

Gestion des erreurs : avec exceptions

Séparation du code de gestion des erreurs du code « normal »

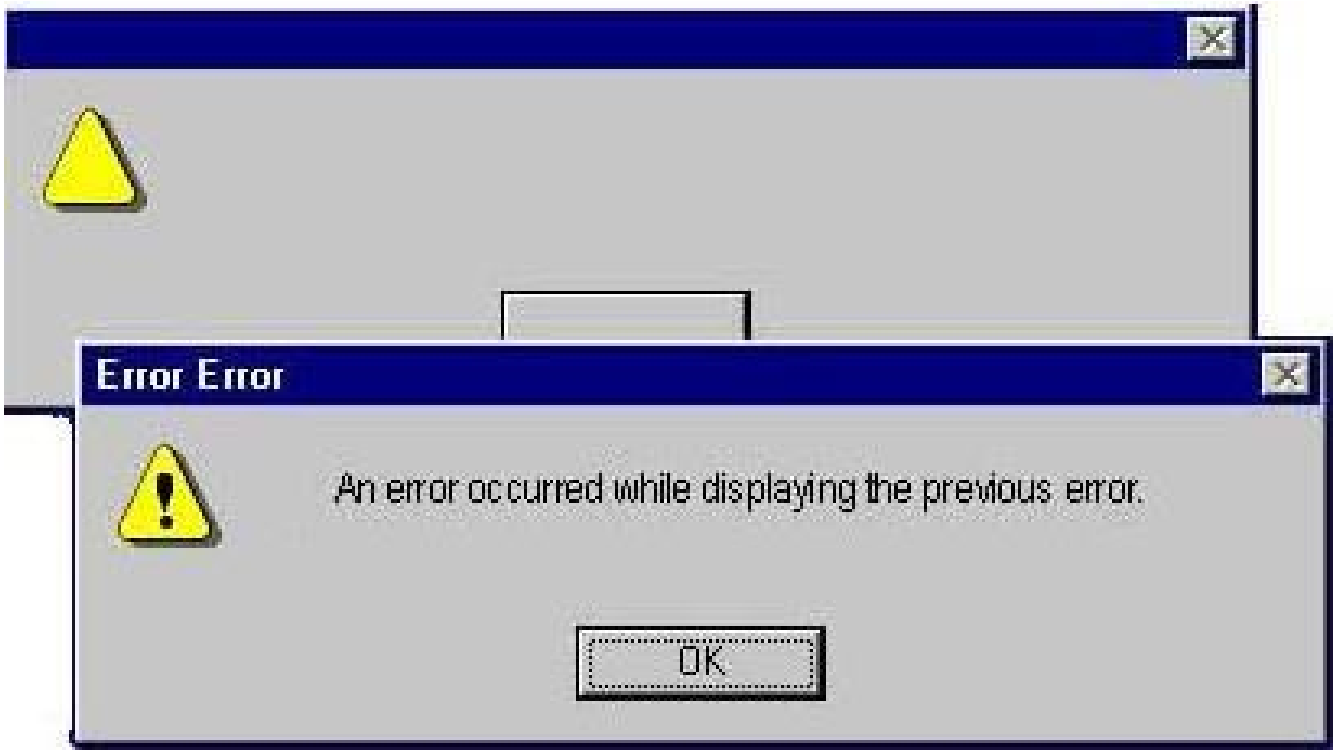
```
readFile {
     try {
          open the file;
          determine its size;
          allocate that much memory;
          read the file into memory;
          close the file;
     } catch (fileOpenFailed) {
          doSomething;
     } catch (sizeDeterminationFailed) {
          doSomething;
     } catch (memoryAllocationFailed) {
          doSomething;
     } catch (readFailed) {
          doSomething;
     } catch (fileCloseFailed) {
          doSomething;
     }
 }
```

- Avantages du mécanisme d 'exceptions
  - *concision*
  - *lisibilité*

Propagation des erreurs

- souvent, on cherche à propager les erreurs dans la pile d'appels
- par exemple le parseur de ligne remonte au parseur de fichier qui remonte à la fonction d'importation...
- mais on ne veut pas perdre la cause de l'erreur
- nécessité de propager les erreurs : facile avec les exceptions qui sont typées
- traitement des erreurs au niveau où c'est le plus approprié

Mais...


Error Error
An error occurred while displaying the previous error.
OK

Conclusion sur les exceptions

- les exceptions rendent la gestion d'erreurs **plus simple et plus lisible**
- le code de gestion des erreurs peut être **regroupé en un seul endroit**
- possibilité de **se concentrer sur l'algorithme** plutôt que de s'inquiéter à chaque instruction des erreurs potentielles
- **les erreurs remontent la pile d'appels** grâce à l'exécutif du langage et non plus grâce à la bonne volonté des programmeurs

Plan

Guide de bonnes pratiques en Java

L'idée derrière cette section du cours est de vous présenter (ou rappeler) un certain nombre de bonnes pratiques Java qui rendront votre code :

- plus lisible
- plus facile à partager avec d'autres développeurs
- plus propre
- plus "sur" (moins de risque d'erreurs bêtes ou d'étourderies)

Un en mot comme en cent

Un code meilleur !
Pour plus d'infos allez voir par exemple :
`http://www.jmdoudoux.fr/java/dej/chap-normes-dev.htm`

Convention de nommage en Java

Rappel des conventions de nommage **à suivre** :

- le nom des classes, des interfaces et des constructeurs commence par une **majuscule** !
- le nom des méthodes et des attributs débute par une **minuscule** !
- pour les méthodes, les attributs et les classes : leurs noms ne peuvent pas contenir d'espaces !
- on utilise la convention `CamelCase` pour les classes : **première lettre en majuscule + à chaque nouveau "mot" dans le nom on met une majuscule** !
- on utilise la convention `lowerCamelCase` pour les méthodes et les attributs : **première lettre en minuscule + à chaque nouveau "mot" dans le nom on met une majuscule** !
- les constantes doivent être écrites en majuscules et chaque mot est séparé par un souligné _ !

Convention plus générales en Java II

Les bonnes pratiques de l'écriture de code en Java recommandent :

- chaque ligne ne doit contenir qu'une seule instruction !
- Si une ligne est longue (plus de 80 caractères) alors vous pouvez la couper en respectant les règles suivantes :
  - couper la ligne après une virgule ou avant un opérateur
  - aligner le début de la nouvelle ligne au début de l'expression coupée
- de toujours utiliser des accolades ouvrantes et fermantes `{ ... }` pour les `if/else`, `for`, etc. Même si ceux ci ne contiennent qu'une seule instruction !
- Les instructions `switch` doivent toutes avoir une clause `default` en fin d'instruction !
- L'instruction `return` ne devrait pas contenir de parenthèses sauf si elles facilitent la compréhension !

Convention plus générales en Java III

Les bonnes pratiques de l'écriture de code en Java recommandent :

- que les blocs `try/catch` soient écrits de la manière suivante :

```
try {
  traitements;
} catch (Exception1 e1) {
  traitements;
} catch (Exception2 e2) {
  traitements;
} finally {
  traitements;
}
```

- Il n'est pas recommandé d'appeler des variables de classe ou des méthodes de classe depuis un objet mais d'utiliser la classe directement !

```
MaClasse.maMethodeStatique();
```

- Si vous souhaitez utiliser des constantes numériques (0, 30, etc.), il est recommandé de déclarer des constantes dans vos classes

Convention plus générales en Java IV

Les bonnes pratiques de l'écriture de code en Java recommandent :

- Pour l'opérateur d'affectation (=), n'utilisez pas d'affectation imbriquée !

```
valeur = (i = j + k ) + m; // pas bien !
i = j + k; // bien
valeur = i + m; // bien
```

- L'usage des parenthèses : il est recommandé d'utiliser des parenthèses lorsqu'il y a plusieurs opérateurs dans une instruction afin d'expliciter la priorité des opérateurs !

```
if (i == j && m == n) { // pas bien !
}
if ( (i == j) && (m == n )) { // bien
}
```

- Il ne faut pas déclarer de variables d'instances ou de variables de classes publiques sans raison valable

Convention plus générales en Java V

Les bonnes pratiques de l'écriture de code en Java recommandent :

- Il est préférable de restreindre l'accès à la variable avec un modificateur d'accès `protected` ou `private` (recommandé) et de déclarer des accesseurs/mutateurs : `getXxx()`, `setXxx()` ou `isXxx()`
- La création de méthodes sur des variables `private` ou `protected` permet d'assurer une protection lors de l'accès à la variable et éventuellement un contrôle lors de la mise à jour de la valeur
- L'écriture de commentaires en Javadoc et dans les méthodes/classes !
- La convention d'indentation à respecter est de 4 espaces par niveau d'imbrication. N'oubliez pas d'utiliser les fonctionnalités de votre IDE préféré pour formater/indenter votre code !

CENTRALE NANTES

Convention plus générales en Java VI

Les bonnes pratiques de l'écriture de code en Java recommandent :

- Essayez de distribuer la complexité de votre code. La complexité dite "**complexité cyclomatique**" se mesure d'après le nombre de choix et de branches dans une méthode. C'est-à-dire le nombre de `if`, `else`, `for`, `while`, `switch/case` dans une méthode
- Il est conseillé d'avoir au maximum une complexité de 12 dans une méthode.
- Plus la complexité d'une méthode est importante et plus elle sera jugée trop complexe et trop dangereuse à modifier. Une modification de ce type de méthodes présente un risque important d'effets de bord et de régressions.
- Il faut éviter d'imbriquer de trop nombreuses instructions/structures de contrôle

Convention plus générales en Java VII

Les bonnes pratiques de l'écriture de code en Java recommandent :

- N'utilisez pas d'espace ni d'accent dans les noms de vos classes, variables, méthodes, fichiers etc.
- Soyez cohérents dans la langue utilisée pour vos commentaires (en général dépend des conventions de la société)
- Les fichiers doivent généralement être encodés en UTF-8

Convention plus générales en Java VIII

Les bonnes pratiques de l'écriture de code en Java recommandent :

- Les fichiers source Java ne doivent pas dépasser 2000 lignes (source : http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141855.html#3043)
- Une ligne ne doit pas dépasser 100 caractères
- Les entreprises possèdent généralement leurs propres conventions que leurs développeurs doivent utiliser, voir par exemple Google : https://google-styleguide.googlecode.com/
- Évitez de dupliquer votre code !! C'est un des pires péchés du développeur
- Rappelez vous : si il y a un bug : Vous êtes le responsable !

Comment mesurer la qualité de son code ?

Il existe des logiciels ou bibliothèques permettant d'évaluer la qualité de votre code (très souvent utilisé en entreprise) :

- PMD : http://pmd.sourceforge.net/
- SonarQube : http://www.sonarqube.org

Sur Internet vous pouvez trouver de nombreuses sources vous donnant une liste des bonnes pratiques :

- http://www.iwombat.com/standards/JavaStyleGuide.html
- https://google-styleguide.googlecode.com/
- http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141855.html#3043

CENTRALE NANTES

Plan

Introduction

- La classe est elle-même un objet
- Son état est représenté par son nom, ses attributs, ses méthodes
- Les méthodes de l'objet classe permettent la manipulation de l'état
  - récupération des attributs, méthodes, constructeurs...
- état de l'objet classe : méta-données stockées dans le byte-code
- à récupérer avec la commande `javap`
- mais aussi manipulation interactive avec le package `java.lang.reflect`

Définitions

- **Introspection** : analyse du sujet par lui-même
- **Réflexivité** : On dit d'un langage de programmation est "réflexif" s'il permet, au moment de l'exécution, d'analyser et d'agir sur le fonctionnement interne du programme lui-même.

Exemple : javap java.lang.Integer I

```
public final class java.lang.Integer extends java.lang.Number
     implements java.lang.Comparable{
    public static final int MIN_VALUE;
    public static final int MAX_VALUE;
    public static final java.lang.Class TYPE;
    static final char[] digits;
    static final char[] DigitTens;
    static final char[] DigitOnes;
    static final int[] sizeTable;
    public static final int SIZE;
    public static java.lang.String toString(int, int);
    public static java.lang.String toHexString(int);
    public static java.lang.String toOctalString(int);
    public static java.lang.String toBinaryString(int);
    public static java.lang.String toString(int);
    static void getChars(int, int, char[]);
    static int stringSize(int);
```

Exemple : javap java.lang.Integer II

```
public static int parseInt(java.lang.String, int) throws
 java.lang.NumberFormatException;
public static int parseInt(java.lang.String) throws
 java.lang.NumberFormatException;
public static java.lang.Integer valueOf(java.lang.String, int)
 throws java.lang.NumberFormatException;
public static java.lang.Integer valueOf(java.lang.String) throws
 java.lang.NumberFormatException;
static void getAndRemoveCacheProperties();
public static java.lang.Integer valueOf(int);
public java.lang.Integer(int);
public java.lang.Integer(java.lang.String) throws
 java.lang.NumberFormatException;
public byte byteValue();
public short shortValue();
```

CENTRALE NANTES

Exemple : javap java.lang.Integer III

```
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
public java.lang.String toString();
public int hashCode();
public boolean equals(java.lang.Object);
public static java.lang.Integer getInteger(java.lang.String);
public static java.lang.Integer getInteger(java.lang.String, int);
public static java.lang.Integer getInteger(java.lang.String,
 java.lang.Integer);
public static java.lang.Integer decode(java.lang.String) throws
 java.lang.NumberFormatException;
public int compareTo(java.lang.Integer);
public static int highestOneBit(int);
```

Exemple : javap java.lang.Integer IV

```
    public static int lowestOneBit(int);
    public static int numberOfLeadingZeros(int);
    public static int numberOfTrailingZeros(int);
    public static int bitCount(int);
    public static int rotateLeft(int, int);
    public static int rotateRight(int, int);
    public static int reverse(int);
    public static int signum(int);
    public static int reverseBytes(int);
    public int compareTo(java.lang.Object);
    ...
}
```

- Ce sont ces méta-données qui permettent à la JVM de générer des exceptions "intéressantes"

Exemple avec “nos classes”

Résultat de javap :

```
javap Personnage
Compiled from "Personnage.java"
public class Personnage {
  protected java.lang.String nom;
  protected int energie;
  protected int ptVie;
  public Personnage(java.lang.String, int, int);
  public void afficheType();
  public java.lang.String getNom();
  public int getEnergie();
  public int getPointsVie();
  public void setNom(java.lang.String);
  public void setEnergie(int);
  public void setPointsVie(int);
  public java.lang.String toString();
  public boolean equals(Personnage);
  public void rencontrer(Personnage);
}
```

La classe `Class`

- `Java` fournit une classe `Class`
- Rappel : tous les objets héritent de la classe `Object` qui possède une méthode `getClass()`
- 2 moyens de récupérer l'objet `Class` associé une à une classe

Objet `Class` associé à la classe `String`

```
public class TestReflectClass {

  public static void main(String[] args) {
    Class c1 = String.class;
    Class c2 = new String().getClass();
  }
}
```

Quelques méthodes de la classe `Class`

- `getSuperClass() : Class` renvoie la super-classe
- `getName() : String` renvoie le nom de la classe
- `getFields() : Field[]` renvoie le tableau des attributs
- `getField(String name) : Field` renvoie le champ `name`
- `getMethods() : Method[]` renvoie la liste des méthodes
- `isInterface() : boolean`
- ...

Intérêt de la méthode `getName` pour nous ?

Surclassement

Nous avons parlé en Cours puis en TP du mécanisme de `Surclassement`. L'idée est de pouvoir "transformer" un objet d'un type en un autre.
Pourquoi on voudrait faire ça ?

Intérêt de la méthode `getName` pour nous ?

Surclassement

Nous avons parlé en Cours puis en TP du mécanisme de `Surclassement`. L'idée est de pouvoir "transformer" un objet d'un type en un autre.
Pourquoi on voudrait faire ça ?

Héritage et listes d'objets de types "différents"

Dans World of ECN on veut gérer une liste de `Personnages`, comme notre diagramme de classes est bien fait, toutes les différentes classes héritent de la classe `Personnage` donc je peux manipuler des objets de type : `List<Personnage>`

- On a donc surclassé les "classes jouables" vers `Personnage` !
- Et si je veux faire faire quelque chose uniquement aux `Guerrier` (ça doit vous rappeler votre TP) ?

Intérêt de la méthode getName pour nous ?

Transtypage

Il est également possible de transtyper des objets du type surclassé (i.e. Personnage) vers leur type effectif ou réel (i.e. celui avec lequel l'objet a été créé, celui du constructeur new XXX())

Transtypage d'un Personnage vers Guerrier

```
Guerrier monG = new Guerrier("Jack", 100, 150, 4);
Magicien monM = new Magicien("Bill", 100, 100, 2);
List<Personnage> listP = new ArrayList<>(); listP.add(monG);
     listP.add(monM);
// Attention je triche !!
for(Personnage p : listP) {
  if(p instanceof Guerrier) {<-- vérifier est-ce que c'est un type Guerrier
    ((Guerrier)p).frapper(monM);
  } else {
    p.rencontrer(monG);
  }
}
```

Intérêt de la méthode `getName` pour nous ?

Plutôt que `instanceof` : réflexivité

Plutôt que d'utiliser `instanceof` qui demande de connaître le nom de la classe (attention après `instanceof` on n'utilise pas une chaîne de caractères !) on pourrait vouloir écrire une méthode plus générale prenant en paramètre un nom de classe (en tant que `String`)

Réflexivité pour test de classe intrinsèque d'un objet

```
public boolean estInstanceDe(Object o, String nomClasse) {
  return (nomClasse.equals(o.getClass().getName()));
}
```

Intérêt de la méthode `getName` pour nous ?

Transtypage d'un `Personnage` vers `Guerrier` avec réflexivité

```
Guerrier monG = new Guerrier("Jack", 100, 150, 4);
Magicien monM = new Magicien("Bill", 100, 100, 2);
List<Personnage> listP = new ArrayList<>();
listP.add(monG);
listP.add(monM);
// Attention je triche !!
for(Personnage p : listP) {
  if(estInstanceDe(p,"Guerrier")) {
    ((Guerrier)p).frapper(monM);
  }
  else {
    p.rencontrer(monG);
  }
}
```

Recherche d'interfaces dans une classe

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

    // On recupere un objet Class
    Class c = new String().getClass();

    // La methode getInterfaces retourne un tableau de Class
    Class[] faces = c.getInterfaces();

    // Pour voir le nombre d'interfaces
    System.out.println("Il y a " + faces.length + " interfaces
     implementees");

    // On parcourt le tableau d'interfaces
    for (int i = 0; i < faces.length; i++) {
      System.out.println(faces[i]);
    }
  }
}
```

Recherche de méthodes dans une classe

```
import java.lang.reflect.Method;

public class TestReflectMeth {
  public static void main(String[] args) {
    Class c = new String().getClass();
    Method[] m = c.getMethods();

    System.out.println("Il y a " + m.length + " methodes dans cette
     classe");

    // On parcourt le tableau de methodes
    for (int i = 0; i < m.length; i++) {
      System.out.println(m[i]);
    }
  }
}
```

Résultats

Exécution

```
Il y a 3 interfaces implementees
interface java.io.Serializable
interface java.lang.Comparable
interface java.lang.CharSequence


Il y a 72 methodes dans cette classe
public boolean java.lang.String.equals(java.lang.Object)
public java.lang.String java.lang.String.toString()
public int java.lang.String.hashCode()
public int java.lang.String.compareTo(java.lang.Object)
public int java.lang.String.compareTo(java.lang.String)
public int java.lang.String.indexOf(int)
public int java.lang.String.indexOf(java.lang.String,int)
public int java.lang.String.indexOf(java.lang.String)
public int java.lang.String.indexOf(int,int)
public int java.lang.String.length()
public boolean java.lang.String.isEmpty()
public char java.lang.String.charAt(int)
public int java.lang.String.codePointAt(int)
public int java.lang.String.codePointBefore(int)
public int java.lang.String.codePointCount(int,int)
...
```

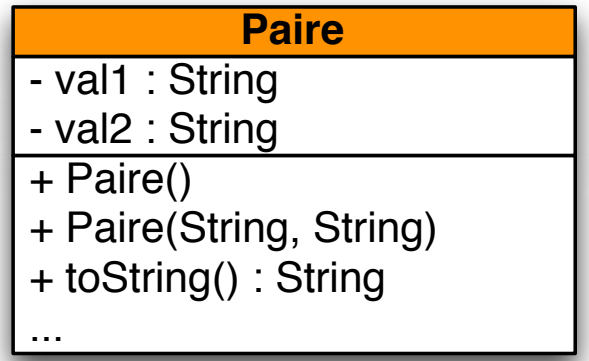
que les méthodes publics

Utilisation

- Inspecter des objets à l'exécution (debugger notamment)
- Instanciation dynamique : créer une instance sans utiliser l'opérateur `new`
  - *Design pattern* `Factory` : instancier/manipuler un objet dont on ne connaît pas le type à la compilation
  - Modifier le comportement d'un programme sans le recompiler **plug-in**
  - Interpréteur de commandes

Instanciation dynamique

- On part d'une classe *Paire*



- `Class.forName(String)` qui renvoie un objet `Class` du type passé en argument
- `Class.newInstance()` qui crée une instance de l'objet `Class`

Instanciation, appel de méthodes dynamique


- Création d'instance sans paramètres

```
String nom = "Paire";
Class clPaire = Class.forName(nom);
Object o1 = clPaire.newInstance();
```

- Création d'instance avec paramètres

```
Class[] types = new Class[] { String.class, String.class};
Constructor ct = clPaire.getConstructor(types);
Object o2 = ct.newInstance((Object[])new String[] {"toto","titi"});
```

- Appel de méthodes

```
Method m = clPaire.getMethod("toString", null);
System.out.println("o1: "+m.invoke(o1,null));
Paire p2 = (Paire) o2;
System.out.println("o2: "+o2);
```

CENTRALE NANTES

Bilan

- Avantages
  - Extensibilité : utilisation de classes écrites par d'autres, sans retoucher au code (plugins)
  - Environnements de développements visuels
  - Debuggers et testeurs de code
- Inconvénients
  - Performances
  - Restriction de sécurité (Applets notamment)
  - Exposition de données internes (rupture de l'encapsulation)