Introduction à la Programmation Objet : Gestion des exceptions

Laboratoire d'Intelligence Artificielle
Faculté I&C

©EPFL 2019
J. Sam

Objectifs du cours de la semaine

- Traitement des erreurs : gestion des exceptions
- Les assertions

©EPFL 2019
J. Sam

Pendant l'heure de cours

- Petit rappel des points importants
- Approfondissements
  - «checked vs unchecked»
  - les assertions

©EPFL 2019
J. Sam

Gestion des erreurs

Les **exceptions** permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple : on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : $1/x$
Si $x = 0$ *erreur* ***Sinon*** *retourner 1/x*

mais que faire **concrètement** en cas d'erreur ?

©EPFL 2019
J. Sam

Gestion des erreurs (2)

① retourner une valeur choisie à l'avance :

```
double f(double x) {
    if (x != 0.0) return 1.0 / x;
    else
        return Double.MAX_VALUE;
}
```

Mais cela

1. **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
2. retourne de toutes façons un **résultat inexact** ...
3. suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

©EPFL 2019
J. Sam
EPFL

Gestion des erreurs (3)

② afficher un message d'erreur
mais que retourner effectivement en cas d'erreur ?...
on retombe en partie sur le cas précédent

```
double f(double x) {
    if (x != 0.0) return 1.0 / x;
    else {
      System.out.println("Erreur dans f :"
                         + "division par 0");
       return Double.MAX_VALUE;
    }
}
```

De plus, cela est **très mauvais** car produit des effets de bord : affichage dans le terminal alors que ce n'est pas du tout dans le rôle de `f`

Pensez par exemple au cas où l'on veut utiliser `f` dans un programme avec une interface graphique... on ne veut alors plus utiliser le terminal mais plutôt ouvrir une fenêtre d'alerte par exemple)

©EPFL 2019
J. Sam
EPFL

Exceptions

Il existe une solution permettant de généraliser et d'assouplir cette dernière solution : déclencher une **exception**

☞ mécanisme permettant de prévoir une erreur à un endroit et de **la gérer à un autre endroit**

Principe :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite donner sur l'erreur
  (« lancer » = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet « lancé »
  (« attraper » = utiliser)
- si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.

Un tel mécanisme s'appelle gestion des exceptions.

©EPFL 2019
J. Sam

Exceptions (2)

Avantages de la gestion des exceptions par rapports aux codes d'erreurs retournés par des fonctions :

- écriture plus facile, plus intuitive et plus lisible
- la propagation de l'exceptions aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...) est fait **automatiquement**
  plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante
- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

(**Note :** si une erreur peut être gérée localement, le faire et ne pas utiliser le mécanisme des exceptions.)

©EPFL 2019
J. Sam
EPFL

Pour résumer

Une exception est un moyen de signaler un événement nécéssitant une attention spéciale au sein d'un programme, comme :

- une **erreur grave**
- une **situation inhabituelle** devant être traitées de façon particulière

☞ but : **améliorer la robustesse des programmes** en :

- séparant le code de traitement des erreurs du code "effectif"
- fournissant le moyen de forcer une réponse à des erreurs particulières

©EPFL 2019
J. Sam

Syntaxe de la gestion des exceptions

On cherche à remplir 4 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif) un moyen de gérer les erreurs qui se présentent
4. éventuellement, "faire le ménage" après un bloc réceptif aux erreurs

On a donc 4 mots du langage Java dédiés à la gestion des exceptions :

`throw`	indique l'erreur (i.e. « lance » l'exception)
`try`	indique un bloc réceptif aux erreurs
`catch`	gère les erreurs associées (i.e. les « intercepte » pour les traiter)
`finally`	(optionel) indique ce qu'il faut faire après un bloc réceptif.

Notez bien que :

- L'indication des erreurs (`throw`) et leur gestion (`try`/`catch`) sont le plus souvent à des endroits bien séparés dans le code
- Chaque bloc `try` possède son/ses `catch` associé(s)

©EPFL 2019
J. Sam

throw

`throw` est l'instruction qui **signale l'erreur** au reste du programme.

Syntaxe : `throw exception`

`exception` est un objet de type `Exception` qui est « lancé » au reste du programme pour être « attrapé »

Exemple :

```
throw new Exception("Quelle erreur !");
```

`Exception` est une classe de `java.lang` qui possède de nombreuses sous-classes et qui descend de `Throwable`.

©EPFL 2019
J. Sam
EPFL

La classe java.lang.Throwable

```
public class Throwable extends Object
```

- Deux constructeurs :
  - Erreur avec ou sans message

```
public Throwable ()
public Throwable (String message)
```

- deux méthodes (parmi d'autres) :
  - Accès au message d'erreur
  - Affichage du chemin vers l'erreur

```
public String getMessage ()
public void printStackTrace ()
```

©EPFL 2019
J. Sam

Les sous-classes de Throwable

Chaque sous-classe décrit une erreur précise

La classe `Error` : 20 sous-classes

- Erreur fatale
- Pas censée être traitée par le programmeur

La classe `Exception` : 121 (hormis les `RuntimeException`)

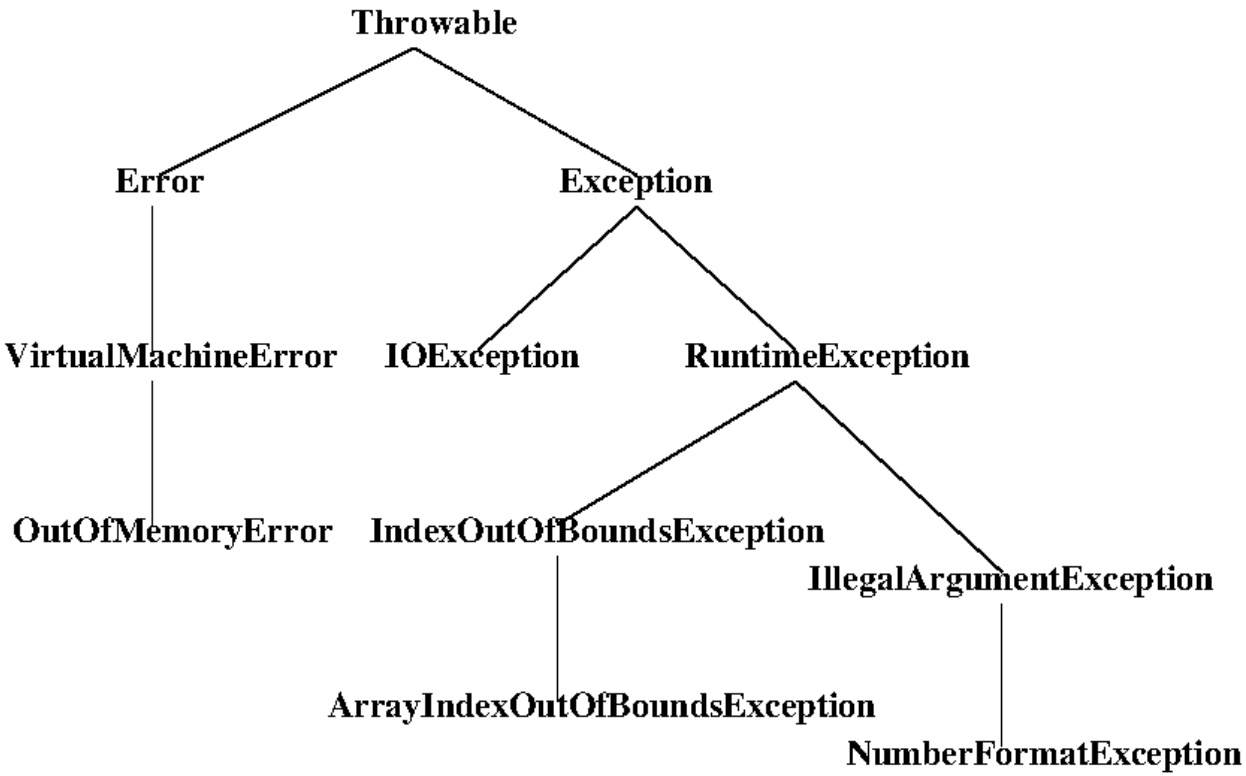
- Circonstance exceptionnelle
- Souvent mais pas toujours une erreur
- Doit être traitée par le programmeur (checked exceptions)

La classe `RuntimeException` : 63 sous-classes

- Exception dont le traitement n'est pas verifié par le compilateur
- Peut être traitée par le programmeur (unchecked exceptions)

©EPFL 2019
J. Sam

Hiérarchie partielle de Throwable



©EPFL 2019
J. Sam

`throw` (2)

`throw`, en « lançant » une exception, interrompt le cours normal d'exécution et :

- saute au bloc `catch` du bloc `try` directement supérieur, si il existe ;
- quitte le programme si l'exécution courante n'était pas dans au moins un bloc `try`.

Exemple :

```
try {
   ...
   // appel contenant un throw Exception
   ...
}
catch (Exception e) {
   ...
}
```

En cas d'erreur, ce code n'est pas exécuté

En cas d'erreur, saute ici

©EPFL 2019
J. Sam

try

`try` (*lit.* « essaye ») introduit un **bloc réceptif aux exceptions** lancées par des instructions, ou des méthodes appelées à l'intérieur de ce bloc (ou même des méthodes appelées par des méthodes appelées par des méthodes... ... ... à l'intérieur de ce bloc)

Exemple :

```
try {
// ...
y = f(x); // f pouvant lancer une exception
// ...
}
```

©EPFL 2019
J. Sam

catch

`catch` est le mot-clé introduisant un **bloc dédié à la gestion** d'une ou plusieurs **exceptions**.

Tout bloc `try` doit toujours être suivi d'au moins un bloc `catch` gérant les exceptions pouvant être lancées dans ce bloc `try`.

Si une exception est lancée mais n'est pas interceptée par le `catch` correspondant, le programme s'arrête (message `Exception in thread ..` et affichage de la trace d'exécution).

Syntaxe :

```
catch (type nom) {
   ...
}
```

intercepte toutes les exceptions de type `type` lancées depuis le bloc `try` précédent
`type` peut-être une classe prédéfinie de la hiérarchie d'exceptions de Java ou une classe d'exception créée par le programmeur.

©EPFL 2019
J. Sam
EPFL

Exemple d'utilisation de `catch`

```
try {
 ...
 if (age >= 150)
    throw new Exception("valeur trop grande");
 ...
 if (x == 0.0)
    throw new ArithmeticException("Division par zero");

}

catch (ArithmeticException e){
   System.out.println(e.getMessage());
   e.printStackTrace();
}

catch (Exception e){
   System.out.println("Qui peut vivre si vieux?");
}
```

☞ toujours intercepter de l'exception la plus spécifique à la plus générale(sinon, erreur signalée par le compilateur)

©EPFL 2019
J. Sam

`catch` (flot d'exécution 1/3)

Un bloc `catch` n'est exécuté **que** si une exception de type correspondant a été lancée depuis le bloc `try` correspondant.

Sinon le bloc `catch` est simplement ignoré.

En l'absence du bloc `finally`, si un bloc `catch` est exécuté, le déroulement continue ensuite normalement **après** ce bloc `catch` (ou après le dernier des blocs `catch` du même bloc `try`, lorsqu'il y en a plusieurs).

En aucun cas l'exécution ne reprend après le `throw` !

©EPFL 2019
J. Sam

`catch` (flot d'exécution 2/3)

Exemple :
en cas d'erreur (lancement d'une exception) :

```
try {
    ...
    // appel contenant un throw Exception
    ...
}
catch (Exception e) {
    ...
    ...
    ...
}
...
```

En cas d'erreur, saute ici

En cas d'erreur, ce code n'est pas exécuté

puis on continue ensuite ici.

©EPFL 2019
J. Sam

`catch` (flot d'exécution 3/3)

Exemple :
si il n'y a pas d'erreur (**pas** de lancement d'exception) :

```
try {
    ...
    // appel contenant un throw Exception
    ...
}
catch (Exception e) {
    ...
    ...
    ...
}
...
```

S'il n'y a *pas* d'erreur, ce code **est** exécuté...

... puis on saute ici

©EPFL 2019
J. Sam

`catch`

Notes :

- Java 7 a introduit le multi-catch : `catch(Exception1 | Exception2 | ..)`
- s'il y a plusieurs blocs catchs toujours les spécifier du plus spécifique au plus général

©EPFL 2019
J. Sam

try/throw/catch dans la même méthode

```
int lireEntier()
{
      try{
       int nbEssais = 0;
       do{
         System.out.println("Donnez un entier:");
         try {
           int i = clavier.nextInt();
           return i;
         }
         catch (InputMismatchException e) {
           System.out.println("Il faut un nombre entier,"
                   + "recommencez!");
           clavier.nextLine();
           ++nbEssais;
         }
       } while(nbEssais <=10);

       throw new Exception("Saisie echouee");
     }
     catch(Exception e){
       System.out.println(e.getMessage());
       System.out.println("fin du programme");
       System.exit(0);
     }
}
```

©EPFL 2019
J. Sam

Le bloc `finally`

Le bloc `finally` est optionnel, il suit les blocs `catch`

Il contient du code destiné à être exécuté qu'une exception ait été lancée ou pas par le bloc `try`

- ☞ But : **faire le ménage** (fermer des fichiers, des connexions etc..)

©EPFL 2019
J. Sam
EPFL

Bloc `finally` : exemple (1)

```
class Inverse {
   public static void main (String[] args) {
      try {
         Integer a = Integer.valueOf(args[0]);
         int b = a.intValue();
         int c = 100/b;
         System.out.println("Inverse * 100 = " + c);
      }
      catch (NumberFormatException e1) {
         System.out.println("Il faut un nombre entier!");
      }
      catch (ArithmeticException e2) {
         System.out.println ("Parti vers l'infini!");
      }
      finally {
        System.out.println("on passe par le bloc final");
      }
   }
}
```

©EPFL 2019
J. Sam

Bloc `finally` : exemple (2)

Exemple d'exécution :

```
>java Inverse 4.1
Il faut un nombre entier!
on passe par le bloc final!!
>java Inverse 0
Parti vers l'infini!
on passe par le bloc final!!
>java Inverse 4
Inverse * 100 = 25
on passe par le bloc final!!
```

©EPFL 2019
J. Sam

« Relancement »

Une exception peut être **partiellement traitée** par un bloc `catch` et attendre un traitement plus complet ultérieur (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de **« relancer » l'exception** au niveau du bloc n'effectuant que le traitement partiel.
(Il faudra bien sûr pour cela que l'appel à ce bloc `catch` soit lui-même dans un autre bloc `try` à un niveau supérieur).

Pour « relancer » une exception, il suffit simplement d'écrire `throw` suivi de l'exception que l'on veut relancer.

Exemple :

```
catch (Exception erreur) {
    // traitement partiel :
   System.out.println("Hmm... pour l'instant  "
                      + "je ne sais pas quoi faire"
                      + "avec l'erreur :"
                      +  erreur.getMessage());
   throw erreur; // relance l'exception capt'ee:
}
```

©EPFL 2019
J. Sam

Exceptions personalisées

Il est possible de programmer **ses propres classes d'exception**

- ☞ sous-classe de `Exception` (ou d'une autre sous-classe d'exception existante)

```
class TropChaud extends Exception
{
   public TropChaud(String message)
       {
          super(message);
       }

   public TropChaud()
       {
          super("La temperature est trop elev'ee";);
       }
}
```

Recommandation : **préserver le comportement de `getMessage`**

On peut mettre tout ce que l'on veut d'autre : codes d'erreur sous la forme d'entiers etc..

©EPFL 2019
J. Sam
EPFL

Déclaration d'une exception

Une méthode lançant une exception sans la traiter localement doit généralement informer qu'elle le fait

Ceci se fait en ajoutant une clause `throws` à l'entête de la méthode
Syntaxe :

```
Type method(..) throws Exception1, Exception2,
..
```

Exemple :

```
int inverse(int x) throws DivisionParZero
{
        if (x == 0)
                throw new DivisionParZero("erreur");
        return 1/x;
}
```

©EPFL 2019
J. Sam

Règle "déclarer-ou-traiter"

Toutes les exceptions en dehors des `RunTimeException` et des `Error` doivent :

- soit **être interceptées** dans la méthode où elles sont lancées ;
- soit **être déclarées** par la méthode.

Si une exception de ce type est lancée sans être interceptée

☞ le compilateur émettra un message d'erreur

☞ «Checked exceptions»

©EPFL 2019
J. Sam

Exemple complet (1)

```
public static void main(String[] args) {
        int nbEssais = 0;
        final int MAX_ESSAIS = 2;
        ArrayList<Double> mesures =
          new ArrayList<Double>();

        scanner = new Scanner(System.in);
        mesures = new ArrayList<Double>();

        do {
           nbEssais++;
                   // remplit le tableau
           acquerirTemp(mesures);
           try {
              plotTempInverse(mesures);
           }
         //...
```

©EPFL 2019
J. Sam

Exemple complet (2)

```
  catch (ArithmeticException e) {
    if (nbEssais < MAX_ESSAIS) {
       System.out.println("Ressaisir les valeurs!");
       restart = true;
    } else {
       System.out.println("Il y a deja eu au moins "
          + MAX_ESSAIS + " essais.");
       System.out.println(" -> abandon");
    }
  }
  }while (nbEssais < MAX_ESSAIS);
}
```

©EPFL 2019
J. Sam

Exemple complet (3)

```
private static void plotTempInverse(
        ArrayList<Double> t) throws ArithmeticException
    throws ArithmeticException
{
        for(int i = 0; i < t.size(); i++) {
            try {
                plot(inverse(t.get(i)));
            } catch (ArithmeticException e) {
                System.out.println("Probleme a l'indice :"
                                   + i);
            // RELANCEMENT
                throw e;
            }
        }
}
```

©EPFL 2019
J. Sam

Exemple complet (4)

```
private static void plot(double x) {

//fait le dessin
}

private static double inverse(double x)
    throws ArithmeticException // PAS NECESSAIRE
                               //RunTimeException
{
    if (x == 0.0) {
        throw new ArithmeticException("Division par 0!");
    }
    return 1.0/x;
}
```

©EPFL 2019
J. Sam

Conseils/mise en garde

La gestion d'une exception coûte beaucoup plus en temps de calcul qu'un simple `if.. then.. else`

☞ **Si l'erreur peut-être traitée là où elle est découverte, il faut le faire sans passer par les exceptions**

Lancer des exceptions spécifiques est plus informatif et utile !

©EPFL 2019
J. Sam

Ce que j'ai appris aujourd'hui

A gérer les erreurs pouvant se produire lors de l'exécution par l'utilisation du mécanisme d'**exception**

- ☞ je peux maintenant rendre mon code plus robuste aux erreurs et aux situations exceptionnelles qu'il peut rencontrer

©EPFL 2019
J. Sam

Les assertions (1)

Les exceptions sont essentiellement le mécanisme par lequel des situations d'**utilisation** "anormales" du programme sont gérées.

Il s'agira donc essentiellement d'erreurs induites par l'utilisateur du programme, telles que :

- fournir le nom d'un fichier inexistant ;
- fournir des données inappropriées ;
- ...

Les erreurs d'implémentation liées au **développement** du programme doivent, par contre, être détectées en amont, pendant les tests systématiques du programme.

☞ **utilisation d'assertions**

©EPFL 2019
J. Sam

Propriété d'un programme

Lors de l'écriture d'un programme on sait souvent que certaines propriétés sont (ou devraient être) vraies, sans que cela ne soit totalement évident à la lecture du code.

Exemple A la fin d'un calcul arithmétique complexe, le résultat doit être dans un intervalle donné.

Connaître ce genre de propriétés à propos d'un morceau de code facilite souvent sa compréhension.

Le fait qu'une propriété qui devrait être vraie ne le soit pas **signale la présence d'un problème**.

- ☞ Il est donc intéressant de pouvoir exprimer ces propriétés

©EPFL 2019
J. Sam
EPFL

Expression des propriétés

Comment exprimer les propriétés (non triviales) que l'on sait être vraies ?

- les décrire dans des commentaires
  - ☞ les commentaires ne sont pas vérifiables automatiquement, ils peuvent être faux et induire en erreur
- faire des tests explicites au moyen d'un if et lever une exception si la propriété n'est pas vraie ☞ le coût du test peut être important, les tests peuvent être redondants car déjà faits à un autre niveau du programme

Altenative : utiliser les **assertions**.

©EPFL 2019
J. Sam

Assertions

Java offre l'énoncé `assert` qui permet d'affirmer qu'une expression booléenne (l'assertion) est toujours vraie.
Par exemple, l'énoncé

```
assert 0.5 <= x && x <= 1.0;
```

est équivalent à :

```
if (!(0.5 <= x && x <= 1.0))
        throw new AssertionError();
```

mais possède deux avantages :

1. il est plus concis,
2. il peut être totalement supprimé au moment du lancement du programme, sans devoir re-compiler quoi que ce soit.

©EPFL 2019
J. Sam

Activation des assertions

La vérification des assertions est **désactivée par défaut**

- ☞ les conditions des assertions ne sont alors **pas évaluées** et ne coûtent rien en temps d'exécution

Pour activer la vérification des assertions, il faut passer l'option `-enableassertions` (ou `-ea`) à la machine virtuelle Java lors du démarrage du programme.

©EPFL 2019
J. Sam

Exemple : recherche dichotomique

La recherche dichotomique peut se programmer au moyen d'une boucle qui réduit à chaque itération l'intervalle de recherche `[l,h]`.

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
    while (l <= h) {
      int m = (l + h) / 2;
      if (a[m] < k) l = m;
      else if (a[m] > k) h = m;
      else return m; // OK, a la position m
        }
    return -1;
    // pas dans le tableau
}
```

©EPFL 2019
J. Sam

Recherche dichotomique : propriété

Une propriété fondamentale de la recherche dichotomique est qu'elle doit **progresser**

- ☞ à chaque itération l'intervalle de recherche doit être (strictement) inclus dans celui de l'itération précédente
- ☞ **Sinon, la recherche ne termine pas** ...

Cette propriété n'est actuellement pas exprimée de façon explicite dans le code de la méthode `binarySearch`.

©EPFL 2019
J. Sam

Recherche dichotomique avec assertions

On peut introduire une assertion exprimant le progrès dans la méthode `binarySearch` :

```
int binarySearch(int[] a, int k) {
    int l = 0, h = a.length - 1;
        while (l <= h) {
                    int m = (l + h) / 2;
                    int l0 = l, h0 = h; //Intervalle d'ava
                    if (a[m] < k) l = m;
                    else if (a[m] > k) h = m;
                    else return m;
                    assert l > l0 || h < h0; //PROGRES
            }
        return -1;
}
```

©EPFL 2019
J. Sam

Recherche dichotomique : tests (1)

On peut tester cette méthode binarySearch en faisant quelques appels.

```
int[] a = new int[] { 2, 3, 5, 7, 11, 17, 19, 23 };
binarySearch(a, 2); // retourne 0, OK
binarySearch(a, 11); // retourne 4, OK
binarySearch(a, 23); // PROBLEME!
```

☞ Le dernier appel provoque la levée d'une exception

```
Exception in thread "main"
java.lang.AssertionError
at binarySearch(BinarySearch.java:8)
```

☞ `binarySearch` est incorrecte !

©EPFL 2019
J. Sam
EPFL

Messages liés aux assertions

Pour mieux comprendre le problème, il est **possible d'ajouter un message** (une chaîne de caractères) à l'assertion, qui est attaché à l'exception :

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
    while (l <= h) {
    int m = (l + h) / 2;
    int l0 = l, h0 = h;
        if (a[m] < k) l = m;
        else if (a[m] > k) h = m;
        else return m;
    assert l > l0 || h < h0: "l="+l+" h="+h; //MESSAGE
        }
        return -1;
}
```

©EPFL 2019
J. Sam

Recherche dichotomique : tests (2)

En reexécutant l'appel problématique :

```
int[] a = new int[] {2, 3, 5, 7, 11, 17, 19, 23 };
binarySearch(a, 23);
```

on obtient maintenant un message d'erreur qui nous permet de comprendre le problème :

```
Exception in thread "main"
java.lang.AssertionError: l=6 h=7
at binarySearch(BinarySearch.java:8)
```

Lorsqu'il n'y a pas plus de deux éléments dans l'intervalle, `m` est égal à `l`.

- ☞ si l'élément recherché n'est pas à la position `l`, le « nouvel » intervalle de recherche est identique au précédent !
- ☞ Plus de progrès !

©EPFL 2019
J. Sam
EPFL

Recherche dichotomique : correction

Ayant compris le problème, on peut maintenant modifier la méthode binarySearch pour qu'elle mette correctement en oeuvre l'algorithme de recherche dichotomique :

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
    while (l <= h) {
    int m = (l + h) / 2;
    int l0 = l, h0 = h;
    if (a[m] < k) l = m + 1; //CORRECTION
    else if (a[m] > k) h = m - 1; //CORRECTION
        else return m;
    assert l > l0 || h < h0;
  }
  return -1;
}
```

©EPFL 2019
J. Sam

Assertions ou exceptions

Les assertions **alertent le développeur** d'un code

☞ elles servent surtout à tester le bon fonctionnement du code

Les exceptions **alertent l'utilisateur** d'un code

☞ elles servent essentiellement à repérer des conditions d'utilisation inappropriées

Lors des assertions, l'exécution du programme s'arrête, tandis que les exceptions peuvent être traitées, par exemple pour redemander un nom de fichier correct.

©EPFL 2019
J. Sam

Assertions ou exceptions

Cependant, **les assertions peuvent être désactivées**, pas les exceptions.

Donc si une vérification de paramètre est très chère, on peut utiliser une assertion pour la faire.

Exemple : vérifier que le tableau passé à `binarySearch` est bien trié.

©EPFL 2019
J. Sam