Les exceptions

Rémi Forax
forax@univ-mlv.fr

Plan

- java.lang.Object
- Les tableaux
- Chaîne de caractères
- Les wrappers

Les Exceptions

- Mécanisme qui permet de reporter des erreurs vers les méthodes appelantes.
- Problème en C :
  - prévoir une plage de valeurs dans la valeur de retour pour signaler les erreurs.
  - Propager les erreurs “manuellement”
- En Java comme en C++, le mécanisme de remonté d’erreur est gérée par le langage.

Exemple d'exception

- Un exemple simple

```
public class ExceptionExample {
  public static char charAt(char[] array,int index) {
    return array[index];
  }

  public static void main(String[] args) {
    char[] array=args[0].toCharArray();
    charAt(array,0);
  }
}
```

- Lors de l'exécution :

```
C:\eclipse\workspace\java-avancé>java ExceptionExample
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at ExceptionExample.main(ExceptionExample.java:18)
```

Exemple d'exception (suite)

- En reprenant le même exemple :

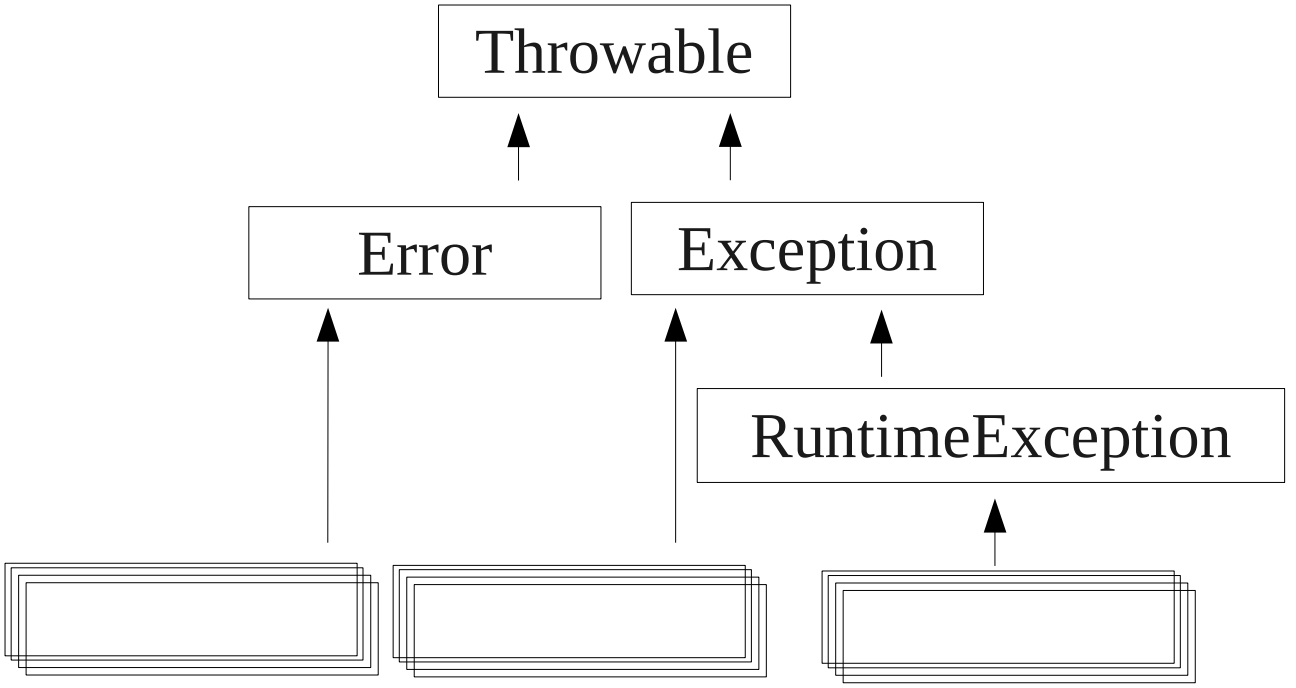
```
public class ExceptionExample {
  public static char charAt(char[] array,int index) {
    if (index<0 || index>array.length)
      throw new IllegalArgumentException("bad index "+index);
    return array[index];
  }

  public static void main(String[] args) {
    char[] array=args[0].toCharArray();
    charAt(array,0);
    charAt(array,1000);
  }
}
```

```
C:\eclipse\workspace\java-avancé>java ExceptionExample toto
Exception in thread "main" java.lang.IllegalArgumentException: bad index
1000
        at ExceptionExample.charAt(ExceptionExample.java:13)
        at ExceptionExample.main(ExceptionExample.java:20)
```

Types d'exceptions

- Il existe 3 trois types d'exceptions organisés comme ceci :



Arbre de sous-typage des exceptions

Types d'exceptions (2)

- Les **Error** correspondent à des exceptions qu'il est rare d'attraper.
- Les **RuntimeException** que l'on peut rattraper mais que l'on n'est pas obligé.
- Les **Exception** que l'on est obligé d'attraper (**try/catch**) ou de dire que la méthode appelante devra s'en occuper (**throws**).

Exceptions levées par la VM

- Les exceptions levées par la VM correspondent :
  - Erreur de compilation ou de lancement
    - NoClassDefFoundError, ClassFormatError
  - problème d'entrée/sortie :
    - IOException, AWTException
  - problème de ressource :
    - OutOfMemoryError, StackOverflowError
  - des erreurs de programmation (runtime)
    - NullPointerException, ArrayIndexOutOfBoundsException, ArithmethicException

Attraper une exception

- try/catch permet d'attraper les exceptions

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=Integer.parseInt(args[0]);
    } catch(NumberFormatException e) {
      value=0;
    }
    System.out.println("value "+value);
  }
}
```

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, exc
character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value
as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) me

Parameters:

s - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

NumberFormatException - if the string does not contain a parsable integer.

Attraper une exception

- try/catch définit obligatoirement un bloc.

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=Integer.parseInt(args[0]);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("no argument");
      e.printStackTrace();
      return;
    } catch(NumberFormatException e) {
      value=0;
    }
    System.out.println("value "+value);
  }
}
```

Sinon cela ne compile pas

Attraper une exception

- Attention, les blocs **catch** sont testés dans l'ordre d'écriture !
- Un catch inatteignable est un erreur

```
public class CatchExceptionExample {
  public static void main(String[] args) {
    int value;
    try {
      value=...
    } catch(Exception e) {
      value=1;
    } catch(IOException e) { // jamais appelé
      value=0;
    }
    System.out.println("value "+value);
  }
}
```

Ne pas attraper tout ce qui bouge

- Comment passer des heures à débugger

```
public static void aRandomThing(String[] args) {
  return Integer.parseInt(args[-1]);
}
public static void main(String[] args) {
  ...
  try {
    aRandomThing(args);
  } catch(Throwable t) {
    // surtout ne rien faire sinon c'est pas drôle
  }
  ...
}
```

- Eviter les catch(Throwable) ou catch(Exception) !!

La directive throws

- Indique qu'une exception peut-être levée dans le code mais que celui-ci ne la gère pas (pas de try/catch).

```
public static void f(String author) throws OhNoException {
  if ("dan brown".equals(author))
    throw new OhNoException("oh no");
}
public static void main(String[] args) {
  try {
    f(args[0]);
  } catch(OhNoException e) {
    tryToRecover();
  }
```

- **throws** est nécessaire que pour les Exception (pas les erreurs ou les runtimes)

Alors **throws** ou **catch**

- Si l'on appel une méthode qui lève une exception non runtime
  - Catch si l'on peut reprendre sur l'erreur et faire quelque chose de cohérent sinon
  - Throws propage l'exception vers celui qui à appelé la méthode qui fera ce qu'il doit faire

Le bloc finally

- Sert à exécuter un code quoi qu'il arrive (fermer un fichier, une connection, libérer une ressources)

```
public class FinallyExceptionExample {
  public static void main(String[] args) {
    ReentrantLock lock = new ReentrantLock();
    lock.lock();
    try {
      doSomething();
    } finally {
      lock.unlock()
    }
  }
}
```

- Le **catch** n'est pas obligatoire.

Exception et StackTrace

- Lors de la création d'une exception, la VM calcule le *StackTrace*
- Le *StackTrace* correspond aux fonctions empilées (dans la pile) lors de la création
- Le calcul du *StackTrace* est quelque chose de couteux en performance.
- Les numéros de ligne ne sont affichées que compilé avec l'option debug
  (pas par défaut avec ant)

Le chaînage des exceptions

- Il est possible d'encapsuler une exception dans une autre
  - **new Exception("msg", throwable)**
  - **new Exception("msg").initCause(throwable)**
- Permet de lever une exception suffisamment précise tout en respectant une signature fixée

Le chaînage des exceptions (2)

- Comme close ne peut pas renvoyer d'**Exception**, on encapsule celle-ci

```
interface Closable {
  public void close() throws IOException;
}
public class DB {
  public void flush() throws OhNoException {
    throw new OhNoException("argh !");
  }
}
public class ExceptionChain implements Closable {
  private final DB db=new DB();
  public void close() throws IOException {
    try {
      db.flush();
    } catch(OhNoException e) {
      throw (IOException)new IOException().initCause(e);
      // ou à partir de la 1.6
      throw new IOException(e);
    } } }
```

Le chaînage des exceptions (3)

- Exécution de l'exemple précédent :

```
Exception in thread "main" java.io.IOException
    at ExceptionChain.close(ExceptionChain.java:27)
    at ExceptionChain.main(ExceptionChain.java:32)
Caused by: fr.umlv.OhNoException: argh !
    at DB.fush(ExceptionChain.java:20)
    at ExceptionChain.close(ExceptionChain.java:25)
    ... 1 more
```

- Désencapsulation :

```
public static void caller() throws OhNoException {
  ExceptionChain chain=new ExceptionChain();
  try {
    chain.close();
  } catch(IOException e) {
    Thowable t=e.getCause();
    if (t instanceof OhNoException)
      throw (OhNoException)t;
    ...
```

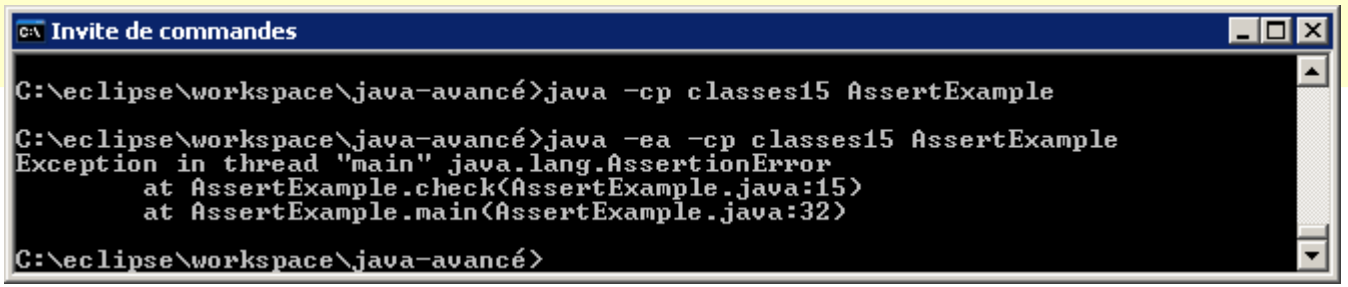
Le mot-clé **assert**

- Le mot-clé **assert** permet de s'assurer que la valeur d'une expression est vraie
- Deux syntaxes :
  - assert *test*; `assert i==j;`
  - assert *test* : *mesg*; `assert i==j:"i not equals to j";`
- Par défaut, les **assert** ne sont pas exécutés, il faut lancer **java -ea** (enable assert)

assert et **AssertionError**

- Si le test booléen du **assert** est faux, la VM lève une exception **AssertionError**

```
public class AssertExample  {
  private static void check(List list)  {
    assert list.isEmpty() || list.indexOf(list.get(0))!=-1;
  }
  public static void main(String[] args) {
    List list=new BadListImpl();
    list.add(3);
    check(list);
    ...
  }
}
```



Exceptions et programmeur

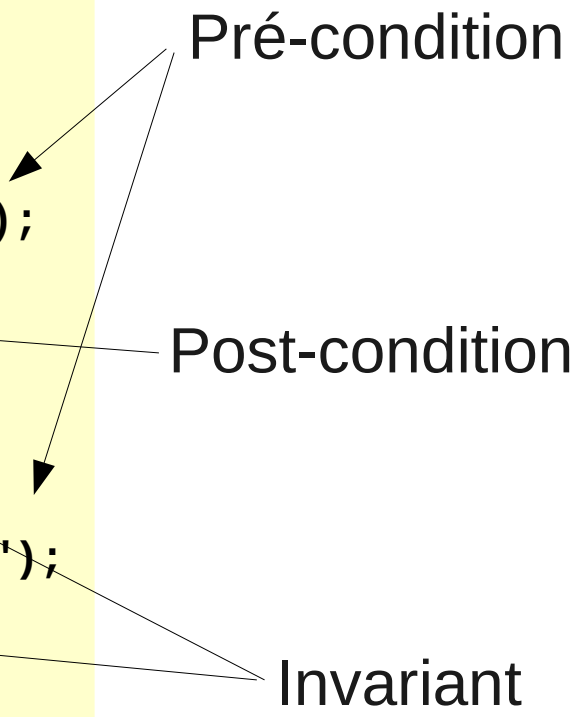
- Le programmeur va utiliser des exceptions pour assurer :
  - Que son code est bien utilisé (pré-condition)
  - Que l'état de l'objet est bon (pré-condition)
  - Que le code fait ce qu'il doit faire (post-condition/invariant)
- Il va de plus gérer toutes les exceptions qui ne sont pas runtime.

Exception et prog. par contrat

- Habituellement, les :
  - Pré-conditions sont utilisées pour :
    - vérifier les paramètres
      NullPointerException et IllegalArgumentException
    - vérifier l'état de l'objet
      IllegalStateException
  - Post-conditions sont utilisées pour :
    - vérifier que les opérations ont bien été effectués
      assert, AssertionError
  - Invariants sont utilisées pour :
    - Vérifier que les invariants de l'algorithme sont préservés. assert, AssertionError

Exemple

```
public class Stack {
  public Stack(int capacity) {
    array=new int[capacity];
  }
  public void push(int value) {
    if (top>=array.length)
      throw new IllegalStateException("stack is full");
    array[top++]=value;
    assert array[top-1]==value;
    assert top>=0 && top<=array.length;
  }
  public int pop() {
    if (top<=0)
      throw new IllegalStateException("stack is empty");
    int value=array[--top];
    assert top>=0 && top<=array.length;
    return value;
  }
  private int top;
  private final int[] array;
}
```



Exemple avec commentaires

- Le code **doit** être commentées

```
public class Stack {
  /** This class implements a fixed size
      stack of integers.
   * @author remi
   */
  public class Stack {
    /** put the value on top of the stack.
     * @param value value to push in the stack.
     * @throws IllegalStateException if the stack
                  is full.
     */
    public void push(int value) {
      ...
    }
    /** remove the value from top of the stack.
     * @return the value on top of the stack.
     * @throws IllegalStateException if the stack
                  is empty.
     */
    public int pop() {
```

Utilisation de Javadoc

```
C:\java-avancé>javadoc src\Stack.java
Loading source file src\Stack.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0-beta3
Building tree for all the packages and classes...
Generating Stack.html...
Generating package-frame.html...
...
```

Class Stack

```
java.lang.Object
  └ Stack
```

```
public class Stack
extends java.lang.Object
```

This class implements a fixed size stack of integers.

Constructor Summary

Stack(int capacity) create a stack with a fixed capacity.	

Method Summary

int	**pop**() remove the value from top of the stack.
void	**push**(int value) put the value on top of the stack.

push

```
public void push(int value)
```

put the value on top of the stack.

Parameters:

value - value to push in the stack.

Throws:

java.lang.IllegalStateException - if the stack is full.

Programmation par exception

- Déclencher une exception pour l'attraper juste après est très rarement performant
(la création du stacktrace coûte cher)

```
public class CatchExceptionExample {
  public static int sum1(int[] array) { // 5,4 ns
    int sum=0;
    for(int v:array)
      sum+=v;
    return sum;
  }
  public static int sum2(int[] array) { // 7,2 ns
    int sum=0;
    try {
      for(int i=0;;)
        sum+=array[i++];
    } catch(ArrayIndexOutOfBoundsException e) {
      return sum;
    }
  }
}
```

StrackTrace et optimisation

- Pour des questions de performance, il est possible de pré-créer une exception

```
public class StackTraceExample {
  public static void f() {
    throw new RuntimeException();
  }
  public static void g() {
    if (exception==null)
      exception=new RuntimeException();
    throw exception;
  }
  private static RuntimeException exception;
  public static void main(String[] args) {
    f(); // 100 000 appels, moyenne()=3104 ns
    g(); // 100 000 appels, moyenne()=168 ns
  }
}
```