Développement Logiciel
L2-S4

Paquetage et Exceptions

anastasia.bezerianos@lri.fr

Les transparents qui suivent sont inspirés du cours de Rémi Forax (Univ. Marne la Vallée)

(transparents utilisés avec son autorisation)

Paquetage

Paquetage

- Un paquetage (*package*) est un regroupement de classes qui traitent un même concept
- Un paquetage :
  - sert à éviter les classes de même nom
  - doit être déclaré au début de chaque classe
  - correspond à un emplacement sur le disque ou dans un jar

A Package

Déclaration d'un Paquetage

- Déclaration de classe avec paquetage


```
package fr.umlv.jbutcher;

public class Rule{
  ...
}
```

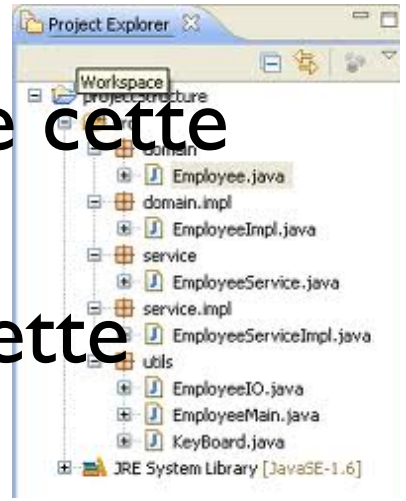
- Ici le nom complet de la classe est **fr.umlv.jbutcher.Rule**
- Règle de nommage:
  fr.masociété.monprojet.monmodule
  ex. com.google.adserver.auth

Organisation sur le disque

- Un paquetage est associé à une structure de dossiers
  ex. **fr.umlv.jbutcher.Rule** à *fr\umlv\jbutcher*)

- Le paquetage (et classes) est disponible à partir de l'adresse
  C:\eclipse\...\jbutcher\src en eclipse

- Soit la compilation s'effectuera à partir de cette adresse
- Soit la variable *CLASSPATH* pointera sur cette adresse



Organisation sur le disque (2)

- On peut aussi utiliser

  javac -d fr\umlv\jbutcher\Rule.java fr\umlv\jbutcher\Rule.class

  pour compiler le paquetage.

- Le paquetage est disponible à partir de "**.**" ou les adresses ajoutés dans la variable *CLASSPATH*
- On peut exécuter une classe de **.** (ou d'un adresse dans *CLASSPATH)* avec son nom complet

  e.g. java fr.umlv.jbutcher.Rule.class

  qui va parcourir la structure fr\umlv\jbutcher

Organisation dans un jar

- Un fichier JAR (Java ARchive) est une collection de classes et de metadonnés, pour distribuer applications ou librairies en java
- On peut avoir des **jar** exécutables avec une classe qui contient la méthode *main()*

- Pour accéder à un jar (e.x. rt.jar), il devra être déclaré dans le *CLASSPATH*

  rt.jar (classes de Java Runtime) en particulier est inclus par défaut dans le boot classpath.

  Au lieu de rt.jar MacOs a le jar classes.jar

Créer un jar

- avec eclipse

  http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/tasks-33.htm

  http://www.cs.utexas.edu/~scottm/cs307/handouts/Eclipse%20Help/jarInEclipse.htm

- ou avec jar

  ```
  C\> jar cvfm MyJar.jar manifest.txt *.class
  ```

  Ici manifest.txt doit declarer la classe avec main(),

  e.x. Main-Class: MyPackage.MyClass

La directive **import**

- La directive import permet d'éviter de nommer une classe avec son paquetage

```
public class MyButcher{
  public static void main(String[] args){
    fr.umlv.jbutcher.Rule rule =
        new fr.umlv.jbutcher.Rule();
  }
}
```

```
import fr.umlv.jbutcher.Rule;
public class MyButcher{
  public static void main(String[] args){
    Rule rule = new Rule();
  }
}
```

- Le compilateur comprend que Rule à pour vrai nom **`fr.umlv.jbutcher.Rule`**
- Le bytecode généré est donc identique

import *

- Indique au compilateur que s'il ne trouve pas une classe, il peut regarder dans les paquetages désignés

```
import java.util.*;
import fr.umlv.jbutcher.*;

public class MyButcher{
  public static void main(String[] args){
        ArrayList list = new ArrayList();
        Rule rule = new Rule;
  }
}
```

- Ici **ArrayList** est associé à **java.util.ArrayList** et **Rule** à **fr.umlv.jbutcher.Rule**

import * et ambiguité

- Si deux paquetages possèdent une classe de même nom et que les deux sont importés en utilisant *, il y a une ambiguité (*Ambiguous import*)

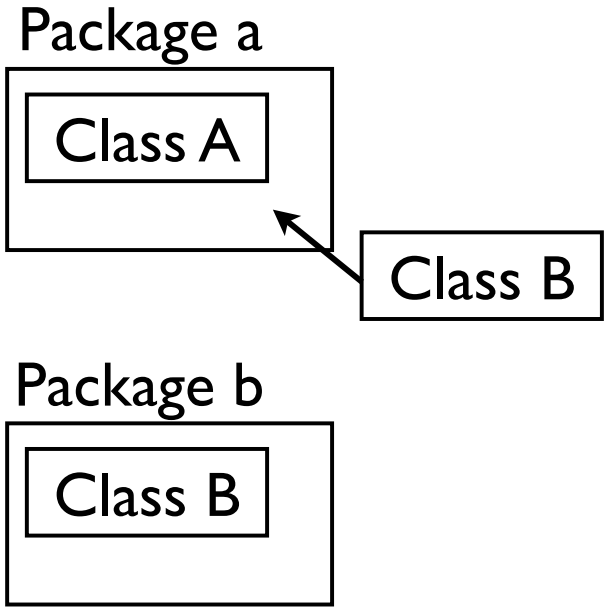
```
import java.util.*;
import java.awt.*;

public class ImportClash {
  public static void main(String[] args){
        List list = ... //oops
  }
}
```

```
import java.util.*;
import java.awt.*;
import java.util.List;

public class ImportClash {
  public static void main(String[] args){
        List list = ... //ok
  }
}
```

import * et maintenance

- **import *** pose un problème de maintenance si des classes peuvent être ajoutées dans les paquetages utilisés

```
import java.a.*;
import java.b.*;

public class ImportClash {
  public static void main(String[] args){
        A a = new A();
        B b = new B();
  }
}
```



- Règle de programmation: éviter d'utiliser des **import ***

import statique

- Permet d'accéder aux membres statiques d'une classe dans une autre sans utiliser la notation "."

```
import java.util.Scanner;
import static java.lang.Math.*;

public class StaticImport {
  public static void main(String[] args){
        Scanner in = new Scanner (System.in);
        System.out.println("Donner un nombre : ");
        double value = sin ( in.nextDouble() );
        System.out.printf("son sinus est %f\n", value)
  }
}
```

- Notation: **import static chemin.classe.*;**

import statique et scope

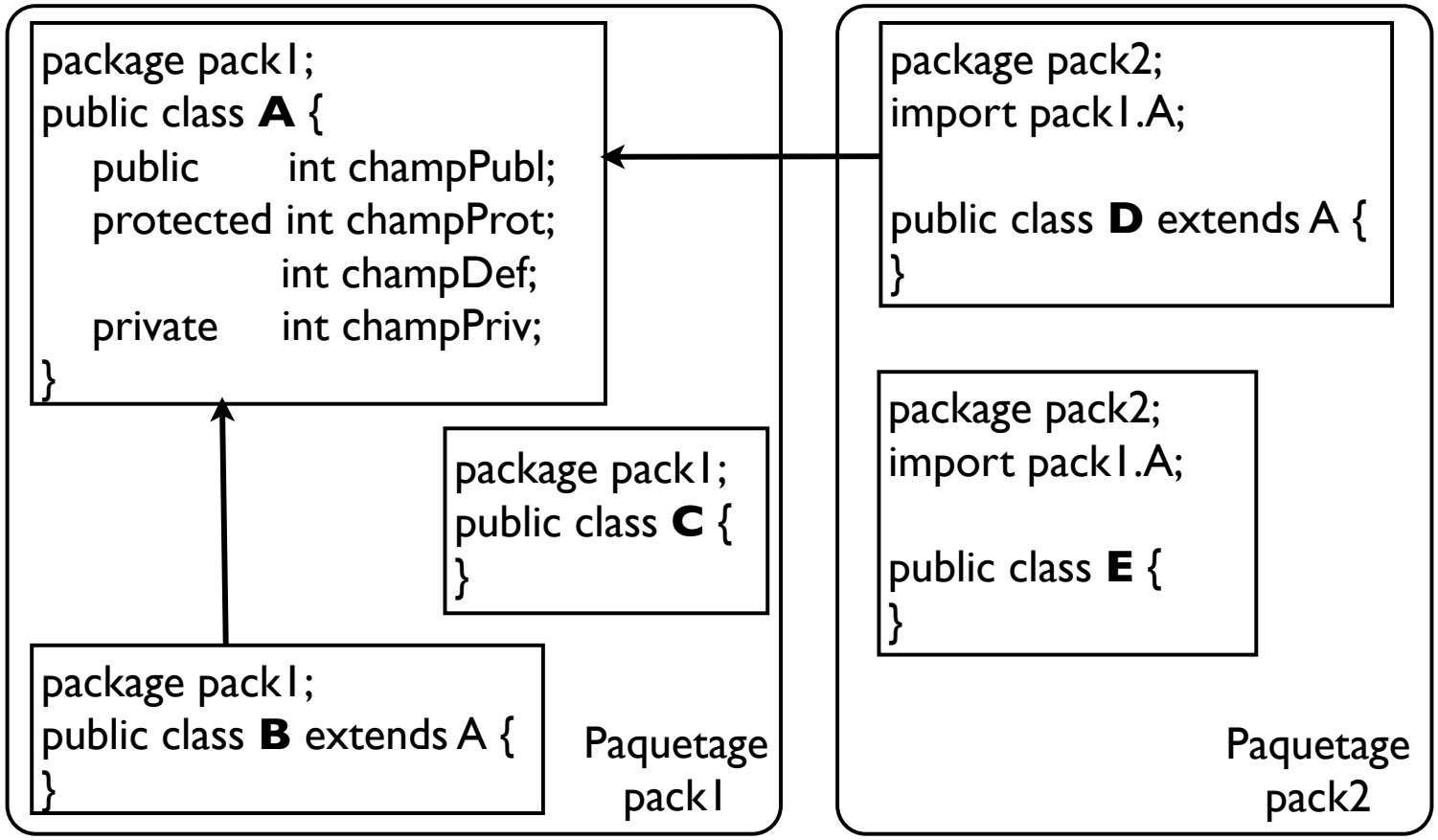
- Lors de la résolution des membres, les membres (même hérités) sont prioritaires sur le scope

```
import static java.util.Arrays.*;

public class WeirdStaticImport {
  public static void main(String[] args){
    java.util.Arrays.toString(args); // ok
    toString(args); // toString() hérité par java.lang.Object
                    // definit comme public String toString();
  }
}
```

- Règle de programmation:

  utiliser l'importa statique avec parcimonie

exemple



	A	B	C	D	E
champPubl	oui	oui	oui	oui	oui
champProt	oui	oui	oui	oui	
champDef	oui	oui	oui		
champPriv	oui				

Exceptions

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\dev-log> java ExceptionExample

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
      at ExceptionExample.main(ExceptionExample.java:18)
```

Exemple d'exception (2)

- En reprenant le même exemple :

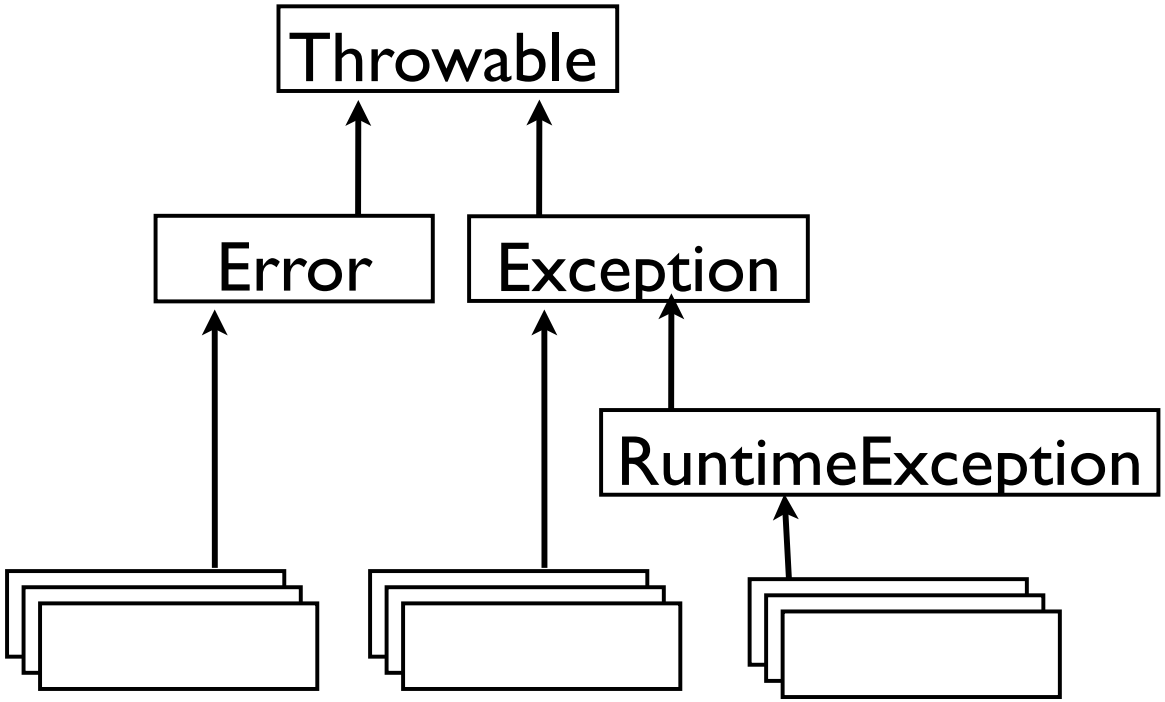
```
public class ExceptionExample {
  public static char charAt (char[] array, int index){
    if (index<0 || index>array.length)
      throw new IllegaleArgumentException("bad index " + index);
    return array[index];
  }
  public static void main(String[] args){
    char[] array = args[0].toCharArray();
    charAt(array,0);
    charAt(array,1000);
  }
}
```

- Lors de l'exécution:

```
C:\eclipse\workspace\dev-log> 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

- Quand une méthode lève une exception (avec **throw**), elle envoie un objet Throwable (ou ses sous-classes)
- Il existe 3 types d'exceptions organisés comme ceci :



Arbre de sous-typage des exceptions

L'arbre pour Java 1.5 http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/package-tree.html)

Types d'exception (2)

- Les **Error** correspondent à des problèmes critiques 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
  - avec (**try/catch**)
  - ou dire avec (**throws**) que la méthode appelante devra s'en occuper

Exceptions levées par la VM

Les exceptions levées par la VM correspondent :

- Erreur de compilation ou de lancement
  - NoClassDefFoundError, ClassFormatError
- problème d'entée/sortie
  - IOException, AWTException
- problème de ressource
  - OutOfMemoryError, StackOverflowError
- des erreurs de programmation (runtime)
  - NullPointerException, ArithmeticException ArrayIndexOutOfBoundsException

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, except that the first character may be an ASCI minus sign `'-'` (`'\u002D'`) to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

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 (2)

- 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 = Integer.parseInt(args[0]);
    } 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) {
    // on trouvera jamais le problème, c'est pas drôle ...
  }
  ...
}
```

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

Attraper une exception (3)

- Quand on attrape une exception (avec un catch), on récupère l'objet d'Exception (par exemple **e**)

```
try {
  ....
} catch (ArithmeticException e) {
    System.out.println ("Impossible de diviser par zéro!");
    // on peut terminer la méthode (ou programme/thread),
    // ou appeler une autre méthode pour continuer le programme
} catch (Exception e) {
    System.out.println ("Une erreur inconnue ... ");
    e.printStackTrace ();
}
```

- Si on ne traite pas une exception le programme termine
- Une méthode utile hérité de Throwable e.*printStackTrace()* imprime la liste des appels effectués avant l'exception (pour debugger).

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);
  } catch (OhNoException e) {
    tryToRecover();
  }
}
```

- **throws** est nécessaire que pour les Exceptions (pas les Erreurs ou les RuntimeExceptions)

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 a appelé la méthode pour faire ce qu'il doit faire

exceptions enchainés

- On peut attraper une exception avec un catch et le passer a la méthode appelante

```
void inner_foo(String author) throws OhNoException {
  if (“dan brown”.equals(author))
    throw new OhNoException (“oh no”);
}
void foo () throws OhNoAgainException{
  try{
    innerfoo (“dan brown”)
  } catch (OhNoException e){
    throw new OhNoAgainException (“oh not again !”, e);
    // ici on rajoute, mais on peut passer la même exception aussi
  }
}
void outer_foo (){
  try{
    foo ();
  } catch (OhNoAgainException e){
    tryToRecover();
  }
-- }
```

Le bloc **finally**

- Sert à exécuter un code quoi qu'il arrive (fermer un fichier, une connection, libérer une ressource) même si on a levé une exception

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

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

Le mot-clé **assert**

- Le mot-clé **assert** permet de s'assurer que la valeur d'une expression est vraie

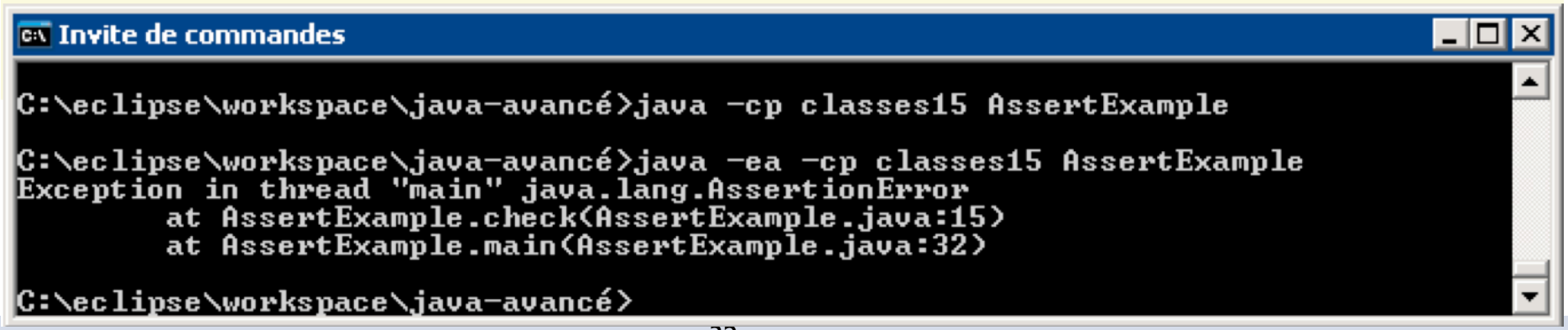
- Deux syntaxes :
  - assert test; `assert i==j;`
  - assert test : mesg; `assert i==j : "i,j not equal";`

- 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 {
  public 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 programmation

- On utilise des exceptions pour assurer:
  - Que notre 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)

- De plus, on gère toutes les exceptions qui ne sont pas runtime

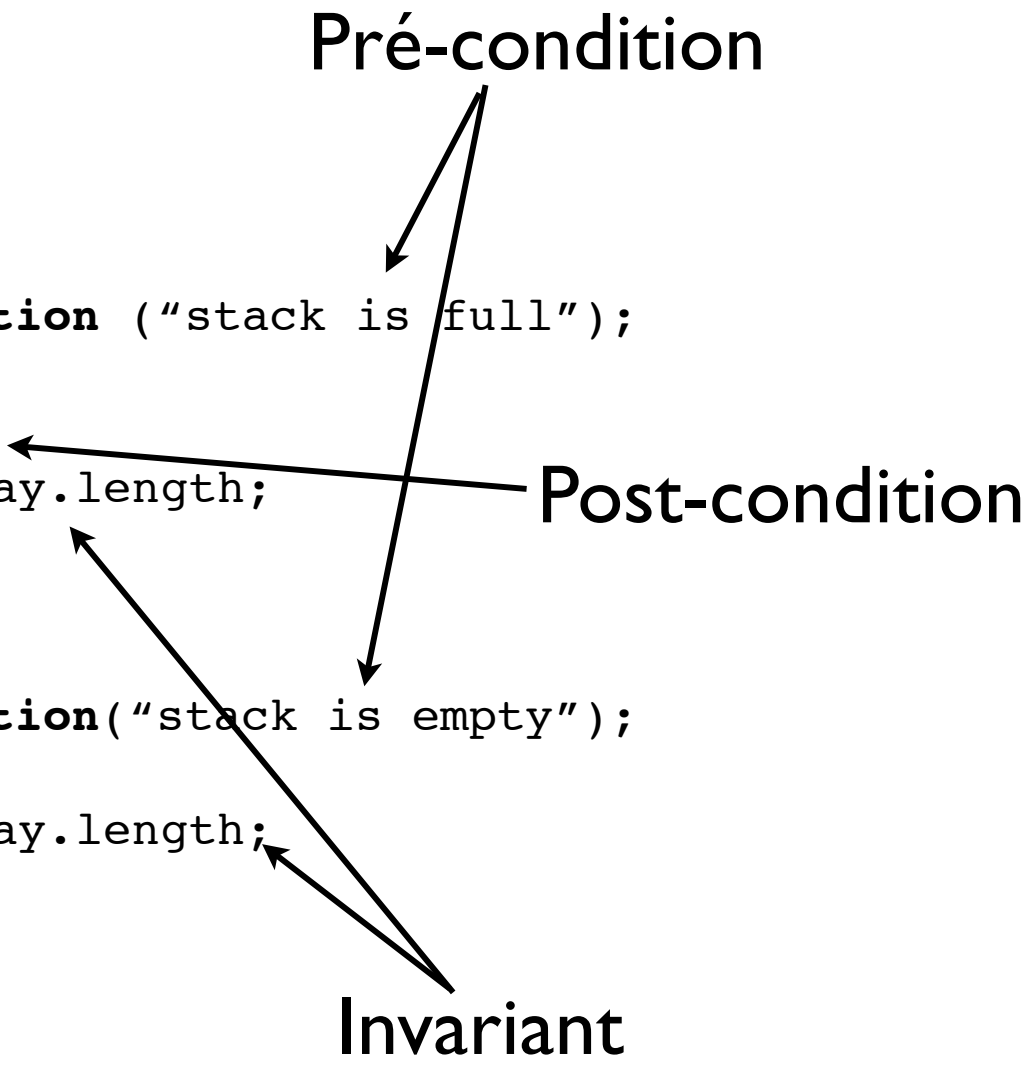
Exceptions et prog. par contrat

Habituellement, les :

- Pré-conditions sont utilisées pour :
  - vérifier les paramètres
    NullPointerException et IlligalArgumentException
  - vérifier l'état de l'objet
    IlligalStateException
- Post-conditions sont utilisées pour :
  - vérifier que les opérations ont bien été effectués
    assert, AssertionError
- Invariants sont utilisées pour :
  - vérifier que les invariantes 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

```
/** This class implements a fixed size stack of integers.
 * @author remi
 */
public class Stack {

  /**
   * Create a stack with a fixed capacity.
   */
  public Stack (int capacity){
    ...
  }

  /** 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){
    ...
  }
  ...
}
```

Utilisation de Javadoc

- ... aussi pour créer un javadoc (avec *javadoc* ou avec eclipse)

```
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.