Sommaire

La fonction factorielle en mathématique

- Définition mathématique :

$$f(n) = \left\{ \begin{array}{ll} 0! = 1 & \text{si } n = 0 \\ n! = n * (n-1)! & \text{si } n > 0 \end{array} \right\}$$

La fonction factorielle en Prolog

- Utilisation du paradigme "logique"
- Implémentation en Prolog :

```
1 /* Factorielle */
2 fact(0,1).
3 fact(X,Y):-X>0 , X1 is X-1 , fact(X1,Z) , Y is Z*X.
```

- L'exemple a été réalisé avec SWI-Prolog (http ://www.swi-prolog.org/)
- Quelques exemples d'utilisation

La fonction factorielle en Prolog

```
#
# debut de trace
#
# Appel de factoriel
1 ?- fact(3,y).
false.
# Un probleme de "matching" ...
2 ?- fact(3,Y).
Y = 6 .
# Nouvel appel et resultat correct
#
1 ?- fact(10,Y).
Y = 3628800 .
#
# fin de trace
```

La fonction factorielle en Prolog

```
#
# debut de trace
#
# On peut aussi generer une serie de resultats avec la
# commande between(de,a,variable)
#
# Lancement de la commande
#
[1] 8 ?- between(1,5,X),fact(X,Y).
X = Y, Y = 1 ;
X = Y, Y = 2 ;
# les premieres valeurs sont bien 1 et 2
# ensuite, cycle normal
X = 3,
Y = 6 ;
X = 4,
Y = 24 ;
X = 5,
Y = 120 ;
# fin de trace
#
```

La fonction factorielle en Prolog

```
# debut de trace
2 ?- trace.
true.
[trace] 6 ?- fact(2,Y).
    Call: (6) fact(2, _G803) ? creep  -> application regle 2. _G803 <- Y
    Call: (7) 2>0 ? creep
    Exit: (7) 2>0 ? creep
    Call: (7) _G878 is 2+ -1 ? creep
    Exit: (7) 1 is 2+ -1 ? creep
    Call: (7) fact(1, _G876) ? creep  -> evaluation de fact(X1,Z) : regle 2
    Call: (8) 1>0 ? creep
    Exit: (8) 1>0 ? creep
    Call: (8) _G881 is 1+ -1 ? creep
    Exit: (8) 0 is 1+ -1 ? creep
    Call: (8) fact(0, _G879) ? creep
    Exit: (8) fact(0, 1) ? creep  -> non evaluable, fin de la recursion
    Call: (8) _G884 is 1*1 ? creep  -> Y = 1 x 1
    Exit: (8) 1 is 1*1 ? creep
    Exit: (7) fact(1, 1) ? creep  -> non evaluable, fin de la recursion
    Call: (7) _G803 is 1*2 ? creep  -> Y = 2
    Exit: (7) 2 is 1*2 ? creep
    Exit: (6) fact(2, 2) ? creep  -> non evaluable, fin de la recursion et fin du programme
Y = 2 .
# fin de trace
```

La fonction factorielle en C

- Paradigmes "fonctionnels" et "impératifs"

```c
#include <stdio.h>
#include <stdlib.h>

/* Version paradigme imperatif */
int factorielle(int n)
{
  int fact;
  int i;
  if(n==0){
    return 1;
  }
  else{
fact = 1;
    for(i=1 ; i <= n ; i++)
    {
      fact=fact*i;
    };
    return fact;
  };
};
```

```c
/* Version paradigme fonctionnel
    */
int factoriellerec(int n)
{
if (n==0) {
  return 1;
} else {
  return (n*factoriellerec(n-1));
}
}

int main()
{
  int n ;
  n=4;
  printf("%ld! vaut %ld \n" , n ,
      factorielle(n));
  printf("%ld! vaut %ld (en
      recursif)\n" , n ,
      factoriellerec(n));
  return 0;
}
```

La fonction factorielle en SCALA

- Paradigme "fonctionnel"

```
package factoriel

object Main {
  def factorial(n:Int):Int = {
   if (n==0)
     return 1
   else
     return n * factorial(n-1)
  }
  def main(args: Array[String]): Unit = {
    println("4! vaut :" + factorial(4))
  }
}
```

La fonction factorielle en JAVA

- Paradigme "objet"

```
public class Nombre {
        int valeur;

        public void setValeur (int n) {
            valeur = n;
        }
        public int getValeur (int n) {
            return(valeur);
        }
        public int calculFact(){
            int res;
            if (valeur <=0) {
                return 1;
            } else {
                res = 1;
                for (int i=1;i<=valeur;i++) {
                    res=res*i;
                }
                return res;
            }
        }
    }
```

La fonction factorielle en JAVA

```
public class Factorielle {

    public static void main(String[] args) {
        Nombre unCalcul = new Nombre();
        unCalcul.setValeur(4);
        System.out.println("4! vaut : \n" + unCalcul.calculFact());
    }
}
```

- A noter que le codage ci-dessus n'est pas une illustration d'une "bonne pratique" en Java mais illustre simplement la vision "objet"

Programmation défensive

- Principes
  - L'idée principale est ici de prémunir son code de l'environnement extérieur
  - Cela signifie prendre des garanties d'une part quant aux données qui sont fournies au code produit et, d'autre part, tenir compte autant que possible des comportements non prévus
  - Plusieurs outils sont disponibles pour le programmeur dont essentiellement les assertions et les exceptions

Programmation défensive : les pré et post-conditions

- L'utilisation des pré et post-conditions est aussi appelée "Programmation par contrat"
- Le programmeur écrit le code du programme en supposant que la pré-condition est assurée
- Le programmeur garantit qu'à l'issue du code la post-condition sera vraie
- Ce principe existe pour de nombreux langages : C, Java, etc. Le premier langage a été Eiffel (Bertrand Meyer - http ://www.eiffel.com/)
- Voici un exemple en Java

Programmation défensive : les pré et post-conditions

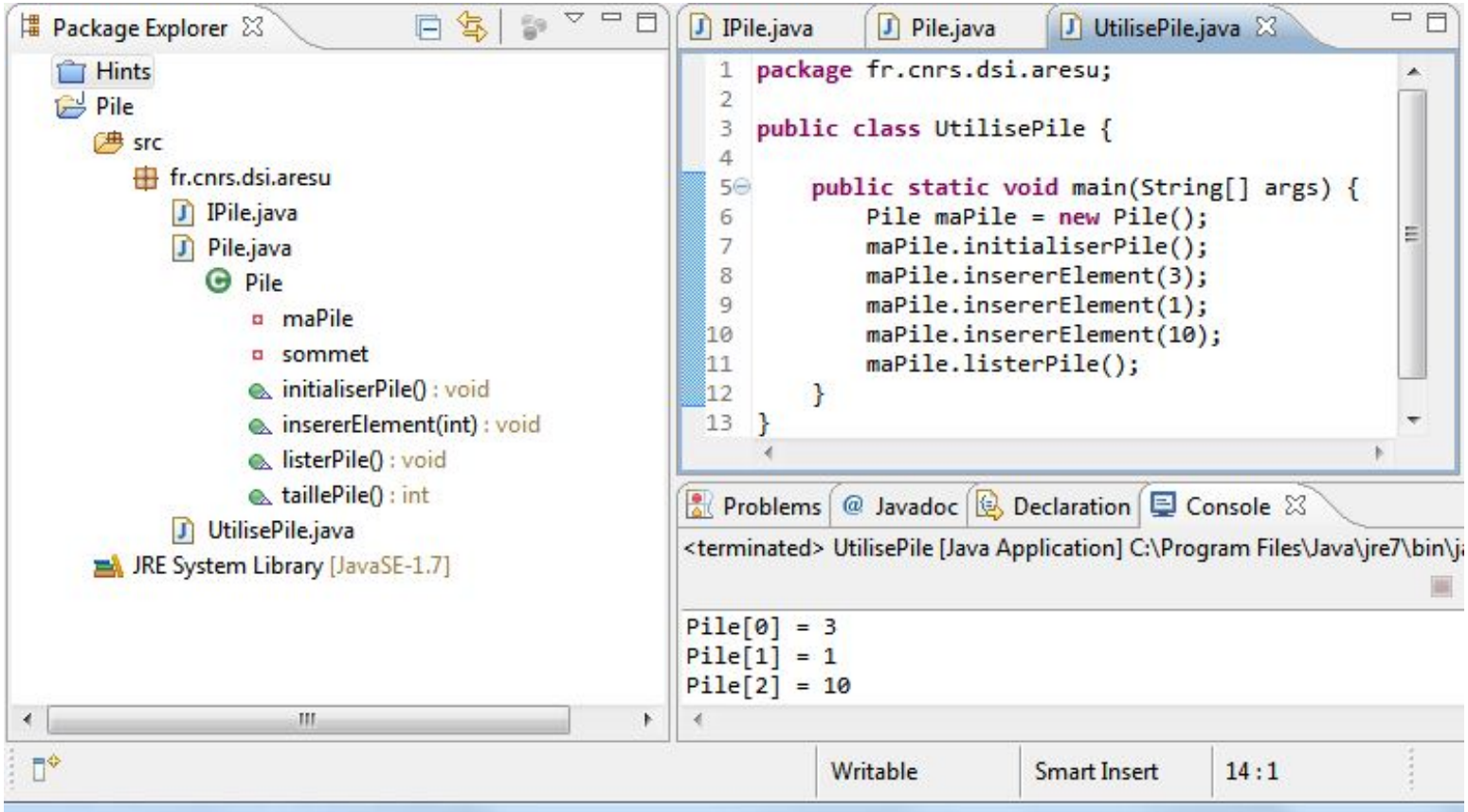


Figure: Le TAD Pile

Programmation défensive : les pré et post-conditions en JAVA

```
public interface IPile {

    public void initialiserPile();
    public int taillePile();
    public void insererElement(int e);
    public void listerPile();
}
```

Programmation défensive : les pré et post-conditions en JAVA

```
public class Pile implements IPile {

    private int[] maPile = new int[5];
    private int sommet = 0;

    public void initialiserPile() {
     for(int i=0 ; i <= (taillePile()-1) ; i++) {
       maPile[i] = 0;
     }
     sommet = 0;
    }
    public int taillePile() return sommet;
    public void insererElement(int e) {
        maPile[sommet]= e;
        sommet = sommet + 1;
    }
    public void listerPile() {
     for(int i=0 ; i < (sommet) ; i++) {
       System.out.println("Pile["+ i + "] = " + maPile[i]);
     }
    }
}
```

Programmation défensive : les pré et post-conditions en JAVA

- Question : que se passe-t-il si on insère 6 éléments ?
- Réponse : on obtient un "java.lang.ArrayIndexOutOfBoundsException" !
- Trois solutions possibles :
  - gérer l'exception avec un "try ... catch" : mais est-ce vraiment un exception, un cas inattendu ?
  - tester le cas en entrée de code : c'est une solution mais pas très élégante
  - gérer par "contrats" les conditions d'exécution du code : là, c'est mieux : −) et plus adapté

Programmation défensive : les pré et post-conditions en JAVA

- La question se pose souvent de savoir quelle solution utiliser entre les "exceptions" et les "contrats"
- La réponse n'est pas toujours simple et est à considérer au regard des éléments suivants :
  - une exception ne se désactive pas contrairement aux assertions !
  - la récupération d'une "exception" permet de peut-être traiter le problème plus loin dans le code
  - une "exception" peut avoir des effets de bord qui peuvent être évités avec des assertions
  - les assertions peuvent s'inspirer de la spécification amont : langage OCL (Object Constraint Language), assertions non exécutables pour UML
- En conclusion :
  - les assertions sont orientées preuves et correction de code
  - les "exceptions" gèrent l'imprévu et améliorent la fiabilité du code
- A noter que ces composantes peuvent/doivent être complétées par une gestion des logs adéquate !

Programmation défensive : les pré et post-conditions en JAVA

- Illustration de l'usage des assertions

```
public void insererElement(int e) {
  assert sommet < maPile.length : "La pile est pleine";
  maPile[sommet]= e;
  sommet = sommet + 1;
}
```

- On obtient alors un message du type : "java.lang.AssertionError : La pile est pleine"
- Le code suivant la directive "assert" n'est pas exécuté

Programmation défensive : les pré et post-conditions en JAVA

- L'utilisation de la directive "assert" est très puissante pour tout ce qui concerne le comportement des méthodes
- Mais cette directive souffre de limites et ne permet pas de faire une vraie spécification par "contrats"
- Java Modeling Language (JML) est un exemple d'extension JAVA pour une programmation par "contrats" plus avancée (développé par Iowa State University - http ://www.jmlspecs.org)
- JML permet de programmer par "contrats" à l'image de ce qui est fait dans Eiffel, avec une logique des prédicats du premier ordre (issue de la méthode axiomatique de HOARE pour les preuves de programmes)
- JML permet :
  - de décrire le comportement des méthodes et des classes
  - de décrire les invariants, pré-conditions et post-conditions
- JML est implémenté :
  - pour effectuer une vérification dynamique : outil JMLRAC par exemple
  - pour effetuer une vérification statique : outil ESC/Java2 par exemple

Programmation défensive : les pré et post-conditions en JAVA

- Google a aussi développé une librairie permettant de programmer par "contrats"
- Cette librairie, cofoja (http ://code.google.com/p/cofoja/) permet d'insérer des annotations dans le code
- Le principe est le même : gestion des pré-conditions, invariants ...
- Voici un exemple avec le TAD Pile

Programmation défensive : les pré et post-conditions en JAVA

```
public class Pile implements IPile {

    private static int MAX_TAILLE = 5;
    private int[] maPile = new int[MAX_TAILLE];
    private int sommet = 0;

....

    @Ensures ({"result <= MAX_TAILLE","result >= 0"})
    public int taillePile() {
       return sommet;
    }

    @Requires ("sommet < MAX_TAILLE")
    @Ensures ("sommet == old(sommet) + 1")
    public void insererElement(int e) {
        maPile[sommet]= e;
        sommet = sommet + 1;
    }
...
}
```

Programmation défensive : les pré et post-conditions en JAVA

```
public class UtilisePile {

    public static void main(String[] args) {
        Pile maPile = new Pile();
        maPile.initialiserPile();
        maPile.insererElement(3);
        maPile.insererElement(1);
        maPile.insererElement(10);
        maPile.insererElement(20);
        maPile.insererElement(100);
        // l'element de trop !
        //maPile.insererElement(8);
        maPile.listerPile();
        maPile.taillePile();
    }
}
```

Programmation défensive : les pré et post-conditions en JAVA

- Dans le premier cas, tout se passe bien et dans le deuxième l'assertion n'est pas vérifiée

```
#
# debut de trace sans l'insertion de l'element 8
Pile[0] = 3
Pile[1] = 1
Pile[2] = 10
Pile[3] = 20
Pile[4] = 100
# fin de trace
#
# debut de trace avec insertion de l'element 8
Exception in thread "main" com.google.java.contract.PreconditionError: sommet < MAX_TAILLE
# fin de trace
```

Programmation défensive : les exceptions en Java

- Les exceptions en Java ont une structure hiérarchique

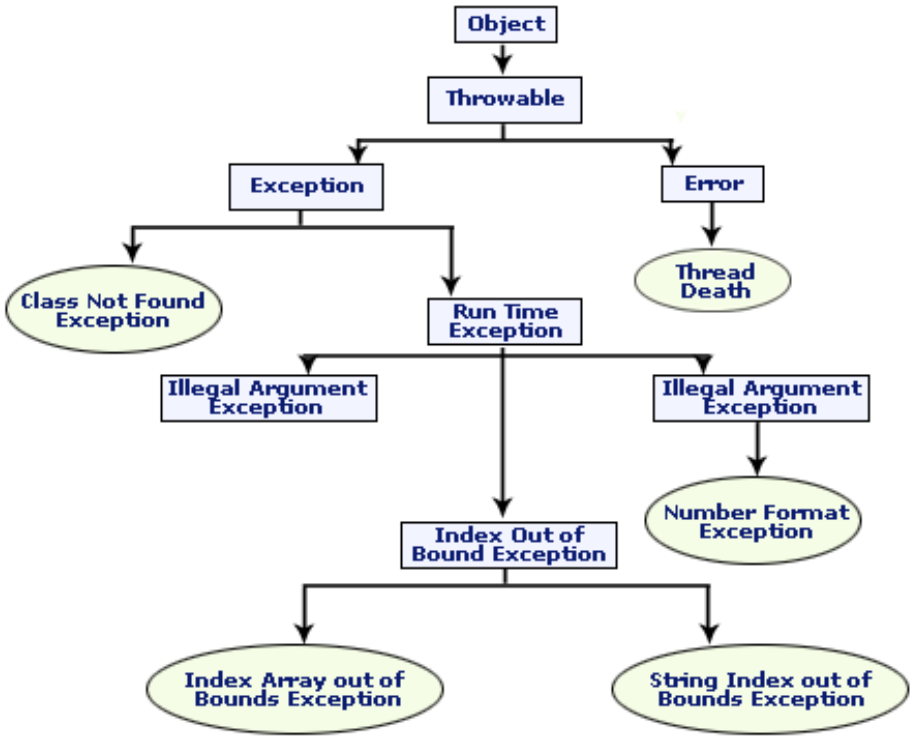


Figure: Exceptions Java (source : rosindia.net)

Programmation défensive : les exceptions en Java

- Écrire sa propre Exception

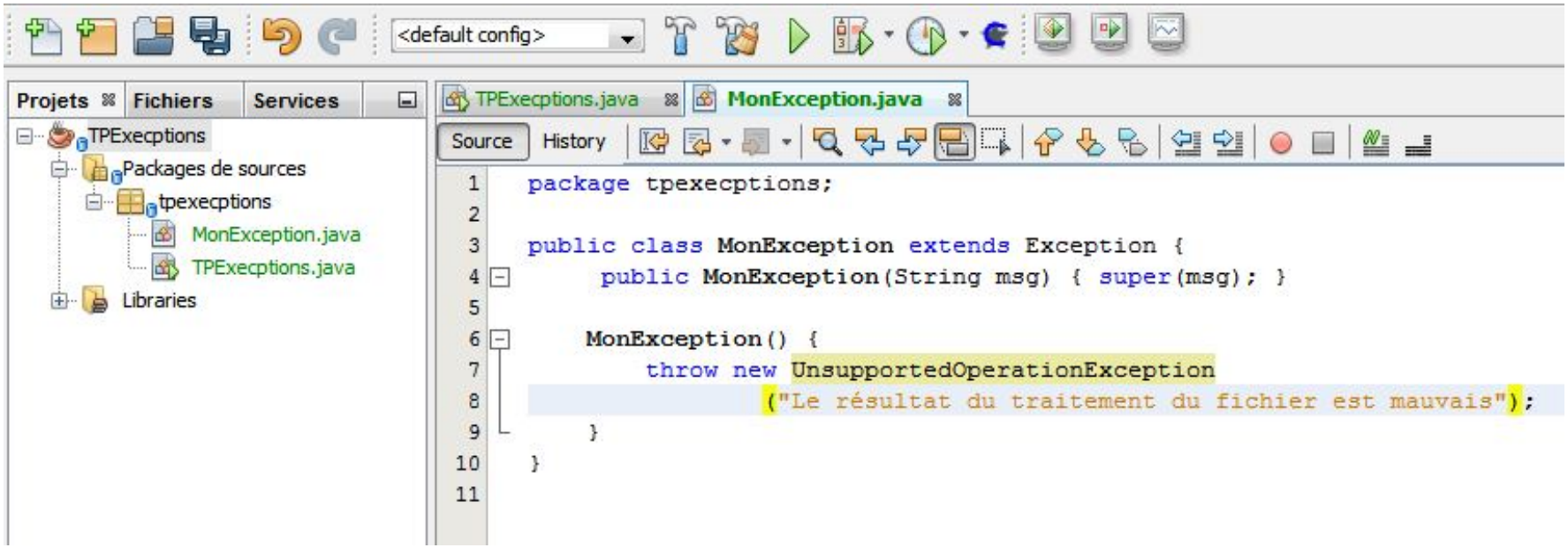


Figure: Exception personnalisée en Java

Programmation défensive : les exceptions en Java

- Gérer les exceptions

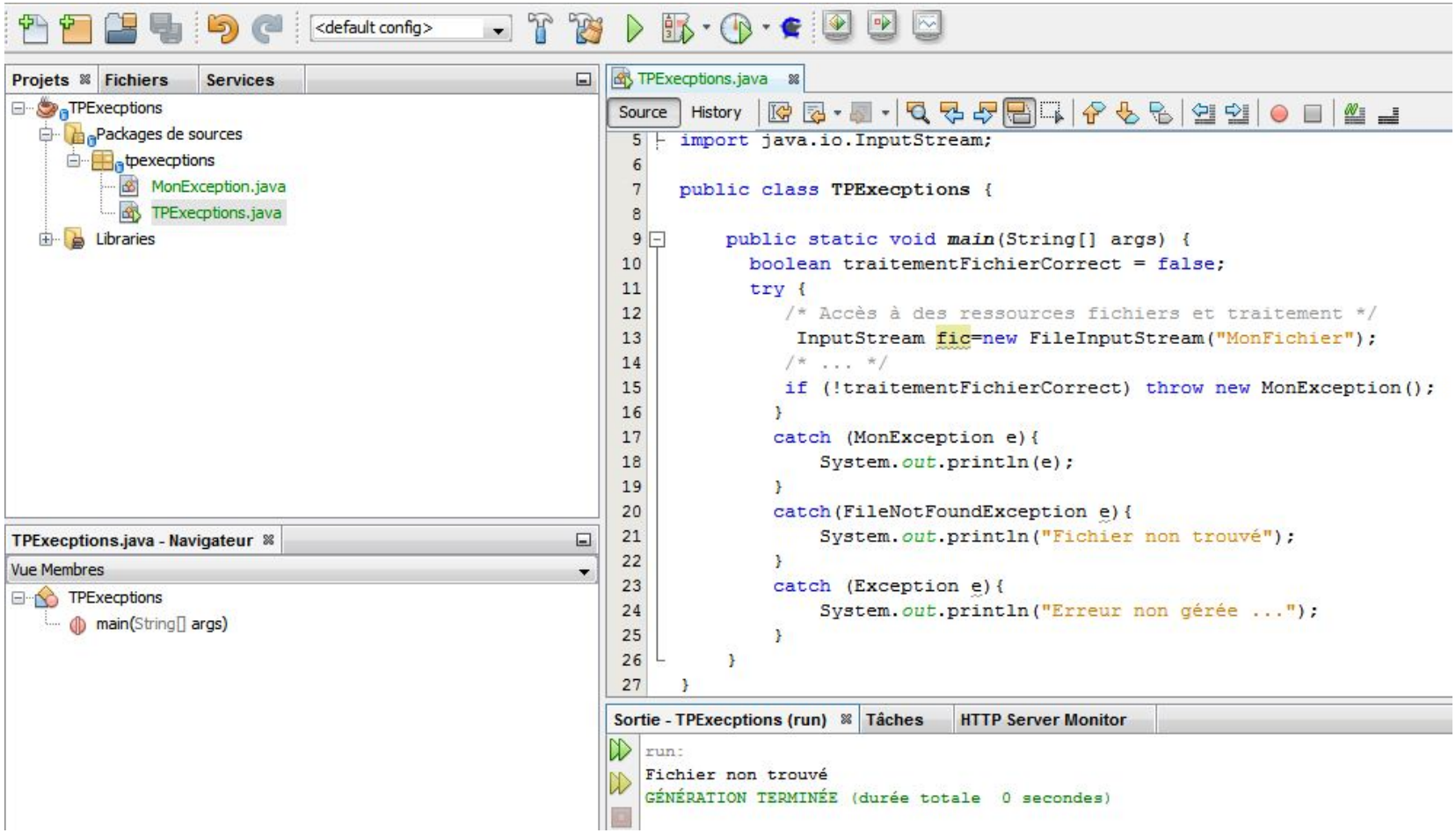


Figure: Gestion des Exceptions Java

Programmation défensive : les exceptions en Java

- La gestion des exceptions en Java mériterait un développement plus important
- C'est un outil puissant de programmation
- Les exceptions permettent de circonscrire une erreur à une portion de code et de la traiter localement
- L'objectif est de "gérer l'imprévu" et de ne pas impacter tout le code (très vrai dans les applications critiques !)

Un mot sur les tests unitaires

- Les tests unitaires doivent être décrits lors de la phase de conception de l'application
- Les tests unitaires ne doivent pas se faire au détriment des tests fonctionnels de recette, d'intégration, de performance, etc.
- Des outils existent pour réaliser des tests : Junit dans l'environnement Java par exemple
- L'idée est de décrire de la manière la plus exhaustive possible le comportement d'une classe
- Parfois, il y a nécessité d'utiliser des objets un peu particuliers (des objets de simulation ou "Mock Objects") qui permettent de répondre en grande partie à la problématique de la simulation d'environnements (BD,WebServices, etc.)
- ... la mise en œuvre n'est pas toujours simple mais fondamentale pour gérer sereinement les évolutions d'un logiciel !

Un mot sur les tests unitaires

```
public class Moteur implements IMoteur {
private String modele;
private String carburant;
private int puissance;

public int consommation() {
return (puissance*100/2);
}
... (getters/setters)
}
```

```
public class MoteurTestTest {

private IMoteur m;

@Before
public void setUp() throws Exception {
IMoteur m = new Moteur();
this.m=m;
}

@Test
public void testTestConsommation() {
m.setPuissance(5);
assertTrue(m.consommation()==250);
}
}
```

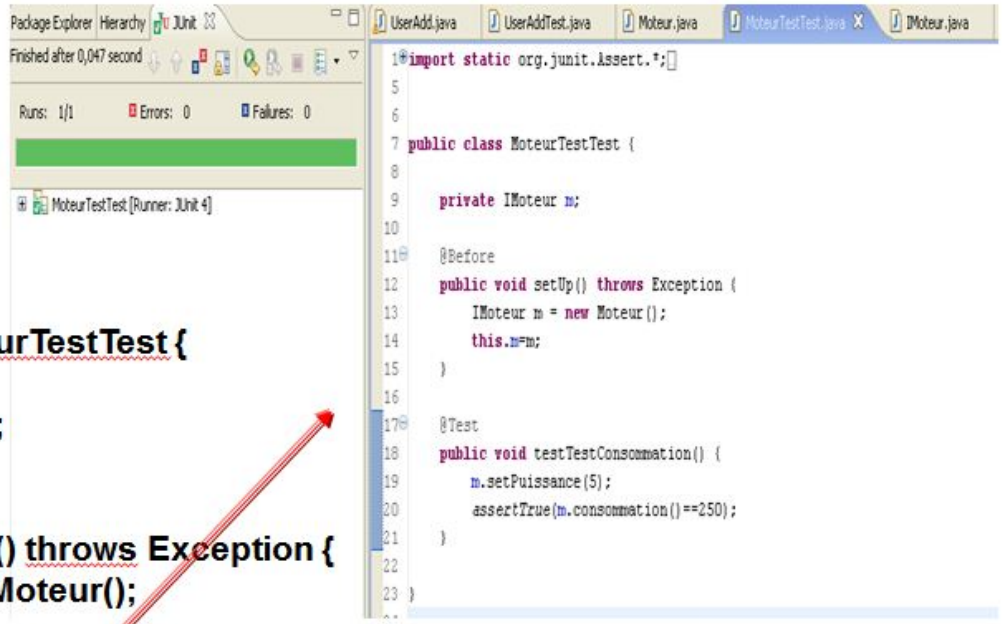


Figure: Test unitaire sous Eclipse

- Voici maintenant 2 "paradigmes" intéressants :
  - la programmation par inversion de contrôle (IoC)
  - la programmation par Aspects
- Cela passe tout d'abord par un petit rappel du pattern "façade" en Java

Le pattern "façade"

- Un "pattern" est un patron de conception
- Un "pattern" est le résultat d'un ensemble de bonnes pratiques en développement et en conception (voir "Gang of Four")
- Il existe beaucoup de "pattern" et sur tous les sujets
- Un "pattern" très utilisé en Java est le "pattern façade"
- Une "façade" est le point d'entrée unique des services d'un sous-système
- Une "façade" permet de séparer la définition d'un système de son implémentation
- En Java, c'est la notion d'interface qui joue ce rôle

Le pattern "façade"

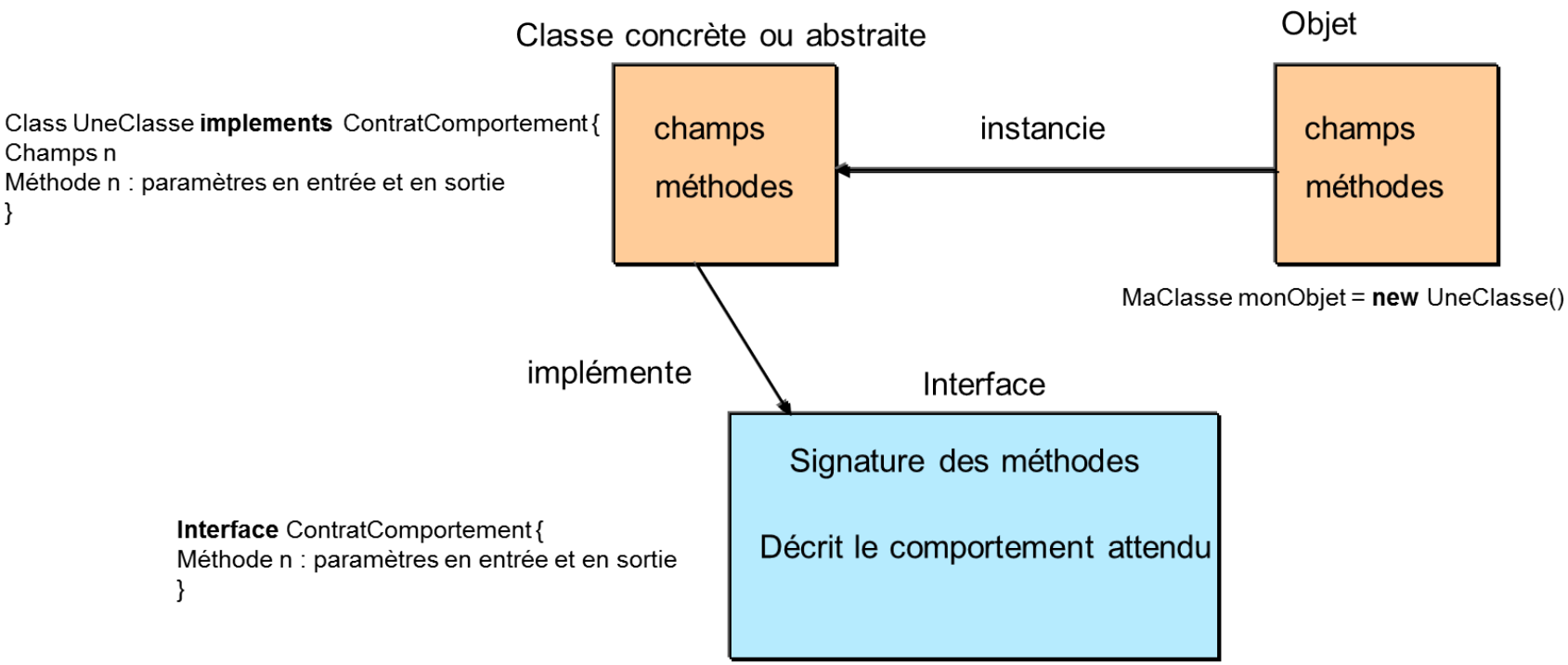


Figure: Patron de conception Façade

Java et la notion d'interface : mise en place du "pattern façade"

- Ce "pattern" est très utile
- Il permet à une couche applicative de ne s'engager que sur la réalisation d'un contrat et non sur son implémentation
- Les couches applicatives bénéficient ainsi d'un niveau d'indépendance supplémentaire
- Par exemple, la couche métier ne connaîtra de la couche d'accès aux données que le nom des méthodes, les paramètres à fournir et le résultat attendu

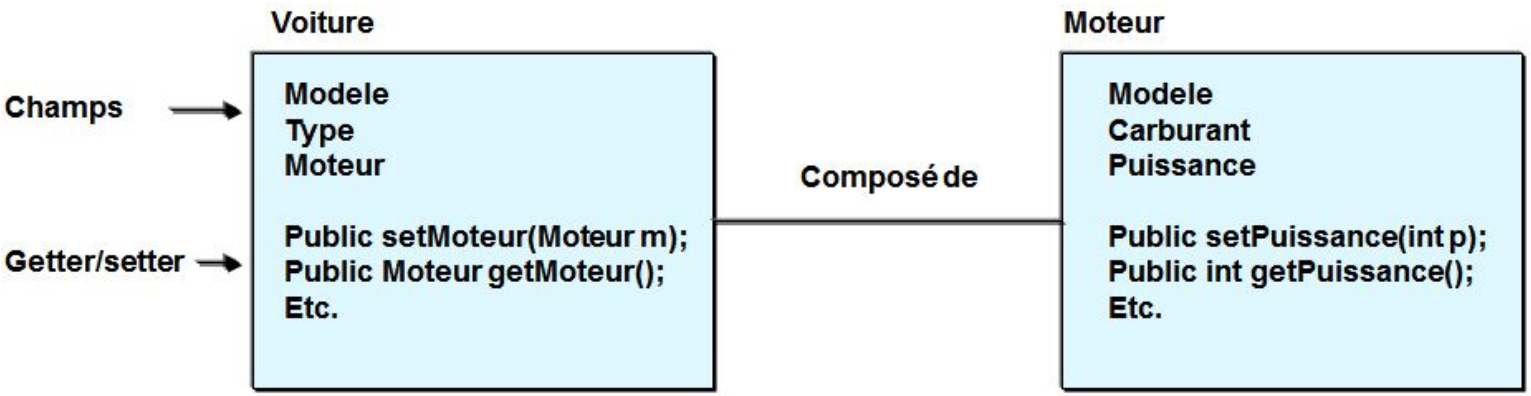
Java et l'IoC

- Les interfaces, c'est bien ... mais cela n'assure tout de même pas une indépendance totale entre couches logicielles ! !
- Une couche métier doit ainsi faire un "new Object()" de la couche DAO ("Data Access Objects")
- Le contrat est respecté grâce aux interfaces mais des objets sont directement référencés entre couches
- Que se passe-t-il en cas de changement de l'objet réalisant l'implémentation ?
- C'est là que la notion d'IoC (Inversion of Control) intervient
- Comparons la programmation "classique" avec le nouveau paradigme d'inversion de contrôle ...

Java et l'IoC

- L'IoC est réputée appliquer le principe d'Hollywood : "Don't call me, I'll call you"
- Pour réaliser cela, il y a nécessité d'utiliser un conteneur
- Il en existe plein ... mais Spring est actuellement un des plus populaires
- Spring est un "framework" offrant les fonctionnalités suivantes :
  - gestion des transactions de façon transversale et centralisée
  - couche d'abstraction à JDBC (facilite les développements JDBC)
  - intégration avec mapping objet-relationnel (Hibernate,...)
  - programmation par Aspects : AOP (Aspect Oriented Programming)
  - framework MVC (Model View Controller), concurrent de Struts
  - ...

Le modèle classique



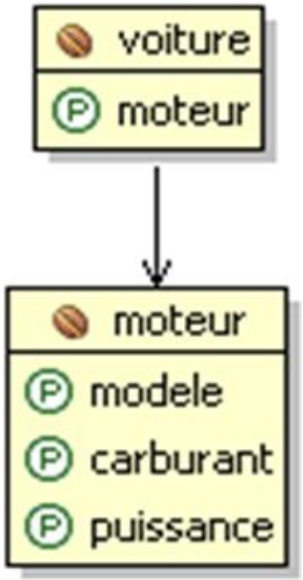
Programmation classique :

```
...
Moteur m = new Moteur();
Voiture v = new Voiture();
v.setMoteur(m);
...
```

Figure: Dépendance entre objets via la création directe d'objets

Le modèle IoC

```
<beans>
  <bean id="moteur" class="Moteur">
    <property name="modele" value="Renault"/>
    <property name="carburant" value="Diesel"/>
    <property name="puissance" value="7"/>
  </bean>
  <bean id="voiture"  class="Voiture">
    <property name="moteur">
      <ref local="moteur" />
    </property>
  </bean>
</beans>
```




```
public class Constructeur {

 public static void main(String[] args) {

  ClassPathResource ressources = new
            ClassPathResource("applicationContext.xml");
  XmlBeanFactory factory = new XmlBeanFactory(ressources);

  Voiture aCar = (Voiture) factory.getBean("voiture");
  System.out.println(aCar.getMoteur().getModele());

  }
}
```

Figure: Dépendance entre objets via IoC

Le modèle IoC

- Vous changez le moteur : il suffit de le déclarer dans le fichier XML, sans impact sur le code ! ! !
- Le moteur va-t-il se comporter comme l'ancien ? Oui, si la notion d'interface est utilisée !
- A noter qu'ici, c'est une déclaration XML qui a été utilisée
- Les notations Java permettent de réaliser les mêmes fonctionnalités

Le modèle IoC

```
public interface IMoteur {

public abstract String getCarburant();
public abstract void setCarburant(String carburant);
public abstract String getModele();
public abstract void setModele(String modele);
public abstract int getPuissance();
public abstract void setPuissance(int puissance);
}
```

```
public class Moteur implements IMoteur {
private String modele;
private String carburant;
private int puissance;

public String getCarburant() {return carburant;}
public void setCarburant(String carburant) {this.carburant = carburant;}
public String getModele() {return modele;}
public void setModele(String modele) {this.modele = modele;}
public int getPuissance() {return puissance;}
public void setPuissance(int puissance) {this.puissance = puissance;}
}
```

Figure: Dépendance entre objets via IoC et "façade"

Le modèle IoC

```
public class Voiture {
private String modele;
private String type;
private IMoteur moteur;

public String getModele() {return modele;}
public void setModele(String modele) {this.modele = modele;}
public IMoteur getMoteur() {return moteur;}
public void setMoteur(IMoteur moteur) {this.moteur = moteur;}
public String getType() {return type;}
public void setType(String type) {this.type = type;}
}
```

Figure: Utilisation de l'interface "Moteur" dans l'objet "Voiture"

- La voiture ne référence désormais que l'interface (//contrat) du moteur
- Aucune adhérence avec l'implémentation du moteur
- Cela facilite la maintenance, les tests, la "parallélisation" des développements, les évolutions logicielles, etc.

En synthèse

- L'IOC est un concept très puissant
- Cela permet de réduire les adhérences entre classes
- Son usage, en plus du pattern "façade" favorise la mise en œuvre des tests unitaires ... et la diffusion du principe de TDD (Test Driven Developpement)

Un aperçu de la programmation par Aspects

- En programmation classique, comment tenir compte des problématiques transversales (gestion des logs, sécurité, transactions, etc.) ?
- Idée : il suffit de rajouter, partout où cela est nécessaire, le/les codes requis pour la fonctionnalité voulue
- Mais plusieurs contraintes apparaissent rapidement : duplication du code, difficultés de maintenance, perte de lisibilité (est-ce que la gestion des transactions doit figurer dans le code métier ?), etc.
- Pour répondre à cette problématique : l'AOP (Aspect Oriented Programming)
- L'idée de base consiste à développer le code sans tenir compte des contraintes transversales et d'ajouter ces contraintes, de manière unique, lors de l'exécution du code
- Ce principe est souvent appelé "Separation of Concerns" ("séparation des préoccupations")

Un aperçu de la programmation par Aspects

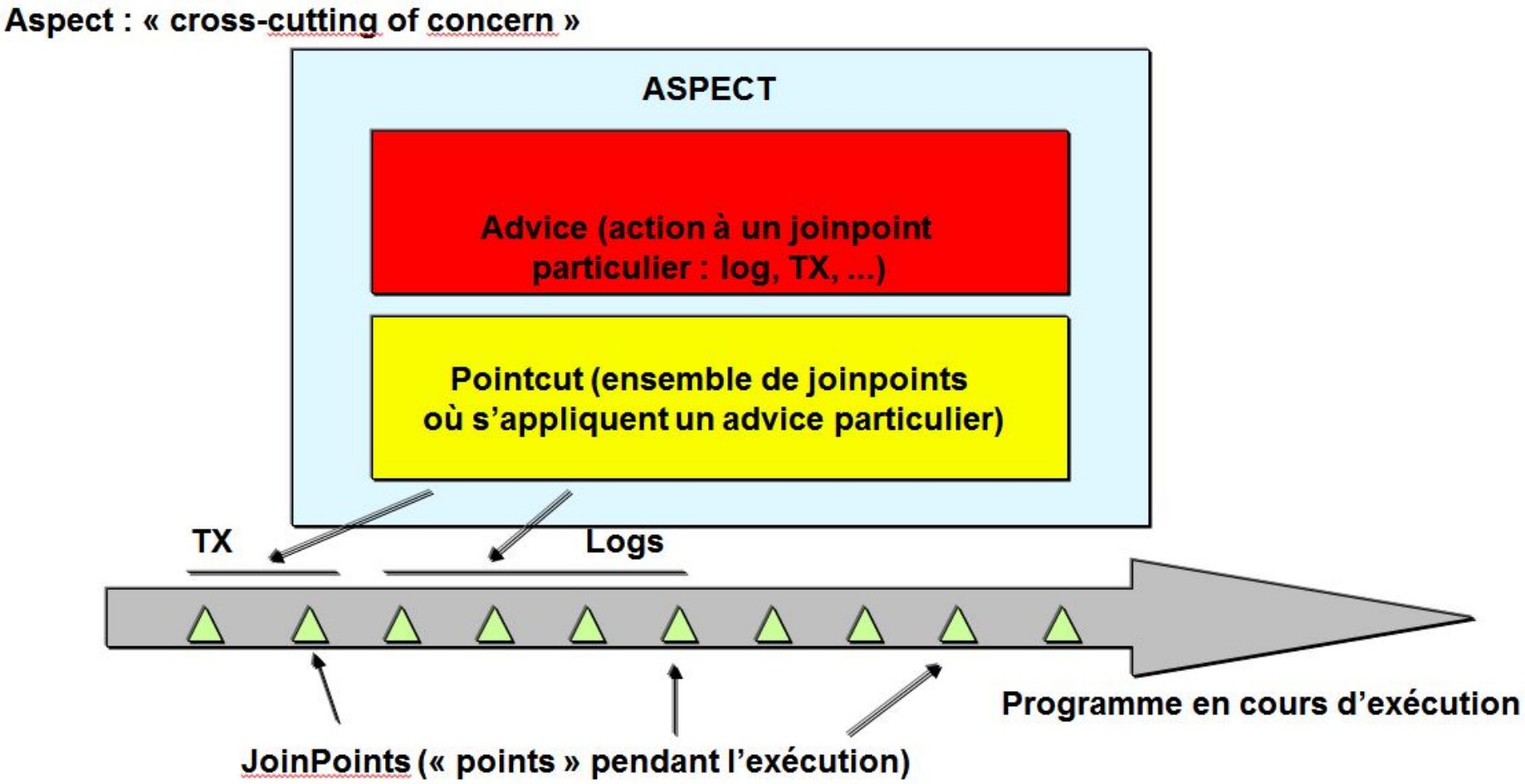


Figure: Joinpoints, Advice et Pointcut

Un aperçu de la programmation par Aspects

- Un aspect est du code Java qui s'applique de manière transparent aux objets

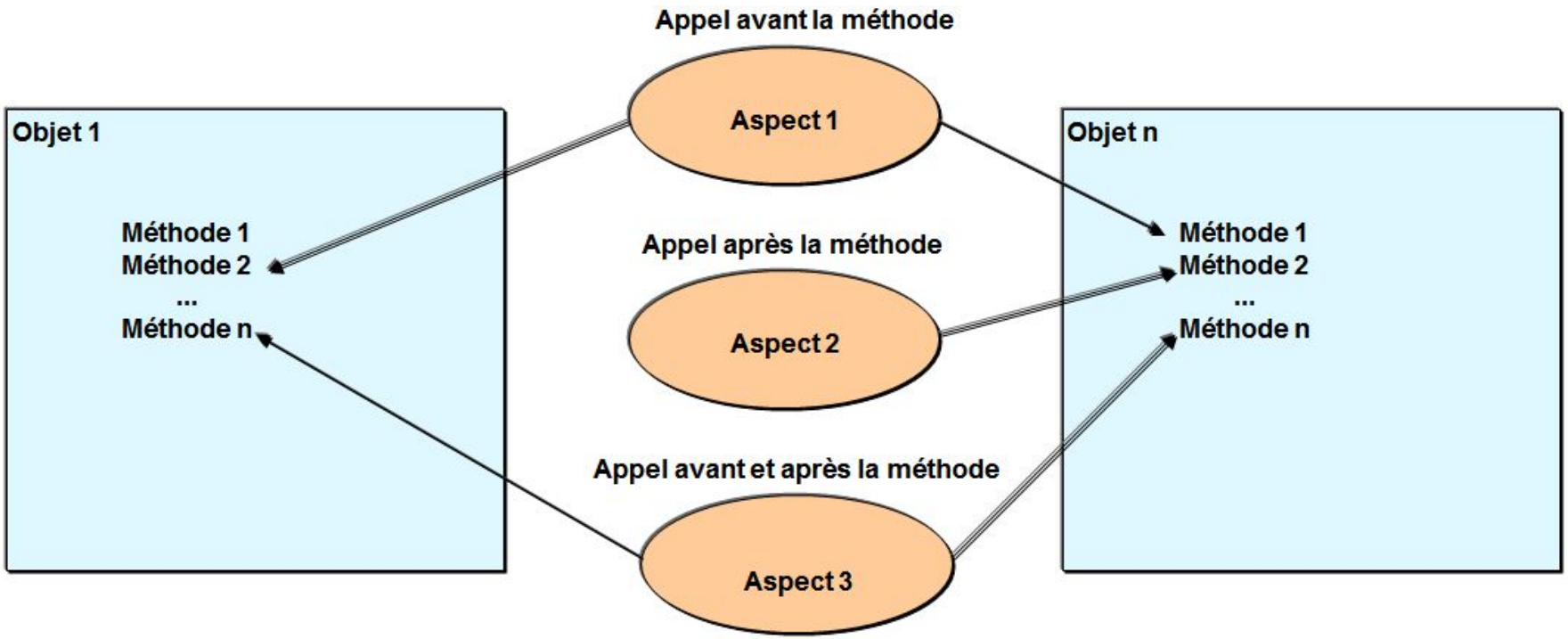


Figure: Méthodes et applications des Aspects

Un exemple d'AOP avec Spring

- 1re étape : créer des objets sans se préoccuper des problématiques transverses
  - on définit le contrat :

```
1  public interface IClassService {
2   public void foo();
3  }
```

  - On l'implémente :

```
1  public class ClassService implements IClassService
2  {
3     public void foo()
4     {
5        System.out.println("Appel de ClassService.foo()");
6     }
7  }
```

Un exemple d'AOP avec Spring

- 2e étape : appeler les objets

```
1  public class MainApplication
2  {
3      public static void main(String [] args)
4      {
5          // Construction du Bean metier
6          IClassService testObject = (IClassService) (new ↩
               XmlBeanFactory(
7                      new ClassPathResource("springconfig.xml"))).↩
                           getBean("classServiceBean");
8          // Execution de la methode publique du Bean metier
9          testObject.foo();
10     }
11 }
```

- on obtient bien : "Appel de ClassService.foo()"

Un exemple d'AOP avec Spring

- 3e étape : on implémente ensuite les problématiques transverses

```
public class TracingBeforeAdvice implements MethodBeforeAdvice {

   public void before(Method m,
                      Object[ ] args,
                      Object target)
                      throws Throwable
   {
      System.out.println(
        "Hello world! depuis " +
        this.getClass().getName() );
   }
}
```

Ce qui doit être fait avant

```
public class TracingAfterAdvice implements AfterReturningAdvice {

   public void afterReturning(Object object,
                              Method m,
                              Object[ ] args,
                              Object target)
                              throws Throwable
   {
      System.out.println(
        "Hello world! depuis " +
        this.getClass().getName() );
   }
}
```

Ce qui doit être fait après

Figure: Implémentation des Aspects

Un exemple d'AOP avec Spring

- 4e étape : on utilise un proxy et on déclare les objets ... le tout dans le fichier de configuration de SPRING

```
<beans>

  <!-- Bean configuration -->
  <bean id="classServiceBean"
  class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>IClassService</value>
    </property>
    <property name="target">
      <ref local="beanTarget"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>theTracingBeforeAdvisor</value>
        <value>theTracingAfterAdvisor</value>
      </list>
      </property>
  </bean>
```

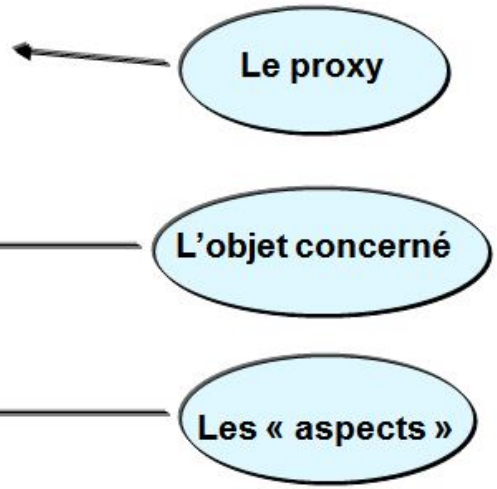


Figure: Configuration XML de Spring

Un exemple d'AOP avec Spring

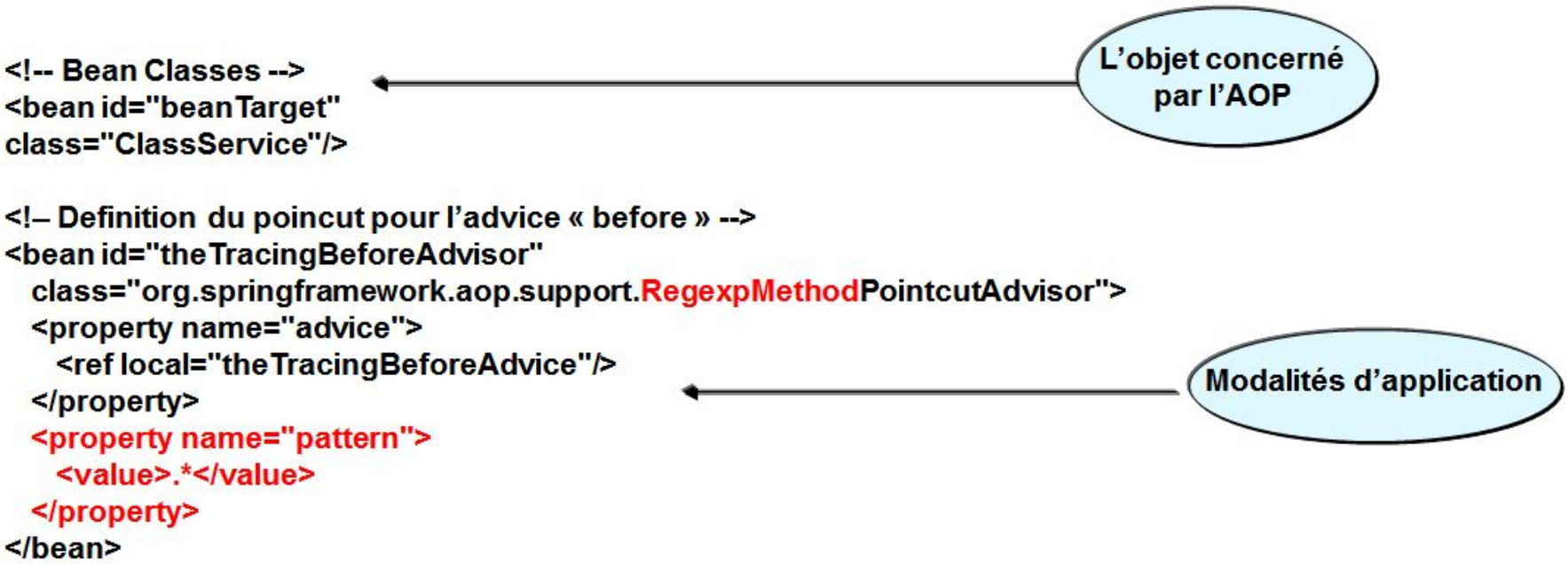


Figure: Configuration XML de Spring

Un exemple d'AOP avec Spring

```
<!-- Definition du poincut pour l'advice « after » -->
  <bean id="theTracingAfterAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="theTracingAfterAdvice"/>
    </property>
    <property name="pattern">
      <value>.*</value>
    </property>
  </bean>

  <!– les classes Advice -->
  <bean id="theTracingBeforeAdvice"
    class="TracingBeforeAdvice"/>
  <bean id="theTracingAfterAdvice"
    class="TracingAfterAdvice"/>

</beans>
```

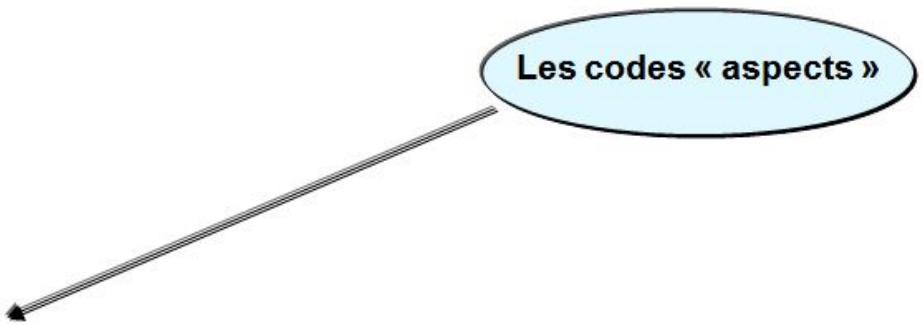


Figure: Principes de l'AOP

Le principe

- Au final, on obtient :

```
Hello world! depuis TracingBeforeAdvice
Appel de ClassService.foo()
Hello world! depuis TracingAfterAdvice
```

Figure: Le résultat final

- Désormais, il facilement possible d'intégrer la gestion des transactions, la sécurité ... et cela sans impacter le code métier ! !
- Remarque : XML est très verbeux ... là encore les annotations Java simplifient la configuration

Spring : les annotations à la rescousse (1/2)

CALLEE :

```
import org.springframework.stereotype.Component;

@Component
public class Callee {
    public String echo() {
        return ("Hello World from Callee");
    }
}
```

CALLER :

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Caller {
    @Autowired
    private Callee b;

    public String echo() {
        return ("Class Caller call Callee : " + b.echo());
    }
}
```

Spring : les annotations à la rescousse (2/2)

```
Configuration XML :
<context:annotation-config/>

<context:component-scan
    base-package="fr.cnrs.dsi.aresu">
</context:component-scan>
```

Sommaire

Quid de l'avenir ?

- Des besoins de plus en plus importants, entre autre pour la recherche
- La fin de la loi de Moore conduit à la mise au point de capacité de calcul répartie (du multicoeur aux grilles de calcul)
- De nouveaux paradigmes de programmation doivent donc émerger et les programmes doivent tirer partie de la puissance et de l'architecture de ces processeurs
- Multithreading, programmation parallèle, etc. mais aussi :
  - la gestion des pannes : le MTBF est de retour !
    - en considérant un MTBF de 1 an pour un processeur : celui-ci diminue considérablement pour 10, 100, 1000, etc. processeurs
    - les codes de programmes doivent le prendre en compte
    - c'est une question rarement évoquée ...
  - la gestion des ressources
    - accès aux ressources partagées : exclusion mutuelle répartie (horloges vectorielles de Lamport et algorithmes suivants)
    - transactionnel réparti

Quid de l'avenir ?

- Évolution des processeurs
  - finesse de la gravure : 22 nm en 2012 (Ivy Bridge), 14 nm en 2015 (Intel Haswell), 10 nm en 2017 (Intel Skymont), 7 nm en 2020 ...
  - Intégration dans le silicium de plus de fonctionnalités : SIMD sur 256 bits (Haswell), permutation de vecteurs, gestion transactionnelle de la mémoire (extension d'AVX2 baptisée TSX dans Haswell), etc.
  - augmentation du nombre de cœurs : de 4 à 8 de Ivy Bridge à Haswell
  - augmentation de la taille des caches (L1, L2 et L3) : exemple Ivy Bridge vers Haswell
    - L1 : 32 Ko à 64 Ko pour les données et les instructions
    - L2 : 256 Ko à 1 Mo
    - L3 : 8 Mo à 32 Mo
    - un L4 pour les GPU ? intégré à la puce ?
  - Ce n'est pas sans conséquences sur les codes (application du principe de "localité" des programmes) et la gestion de la mémoire répartie (problématiques de cohérences de caches répartis : NUMA, etc.)

Quid de l'avenir ?

- Évolution des langages (exemple de Java)
  - Oracle a prévu des évolutions jusqu'en 2021
  - Java 7 (version actuelle), Java 8 (septembre 2013), Java 9 (2015) ... Java 12 (2021)
  - Intégration de plus en plus native de la parallélisation des traitements : "JSR 335 (Lambda Expressions for the JavaTM Programming Language) aims to support programming in a multicore environment by adding closures and related features to the Java language" (source OpenJDK)
  - D'autres évolutions : Jigsaw (JDK modulaire), etc.
  - Actuellement, le JDK 7 améliore déjà la gestion du parallélisme : API ForkJoin issue de la JSR166y
- Globalement, il ne faut pas oublier que :
  - les programmes doivent être parallélisables : les propriétés de convergence restent-elles valables dans le cas du parallélisme ? Les résultats obtenus sont-il associatifs ?
  - la loi d'Amdahl est là pour rappeler le "principe de réalité" (source http ://www.extremetech.com)

Illustration de la loi d'Amdahl

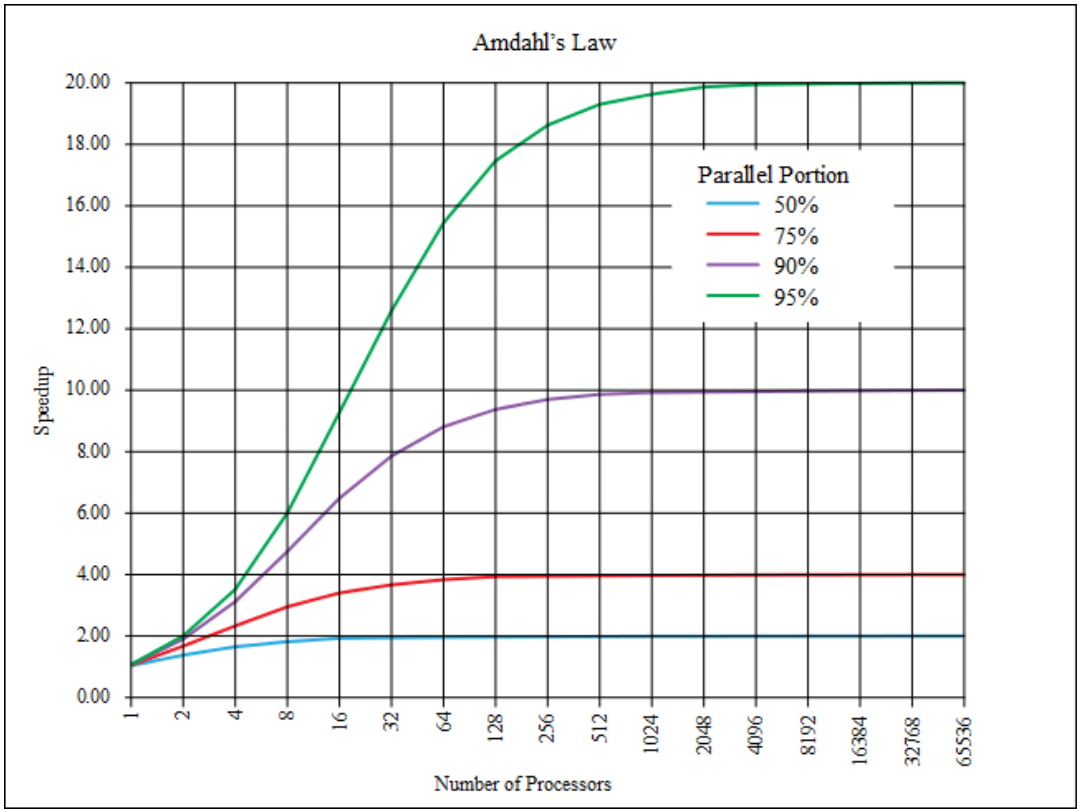


Figure: Loi d'Amdahl

Quid de l'avenir ?

- Il reste donc beaucoup de travail à faire pour exploiter au maximum l'augmentation du nombre de cœurs
- Il faut, entre autre :
  - augmenter le pourcentage de code parallélisable dans les programmes
  - appréhender différemment l'accès aux ressources (gestion plus fine des "sections critiques")
- Des solutions émergent sous forme de "Framework" pour appréhender le traitement de grande volumétries de données
- Une des exemples significatifs est "MapReduce" (Google) et une de ses implémentation Hadoop de la fondation Apache

MapReduce

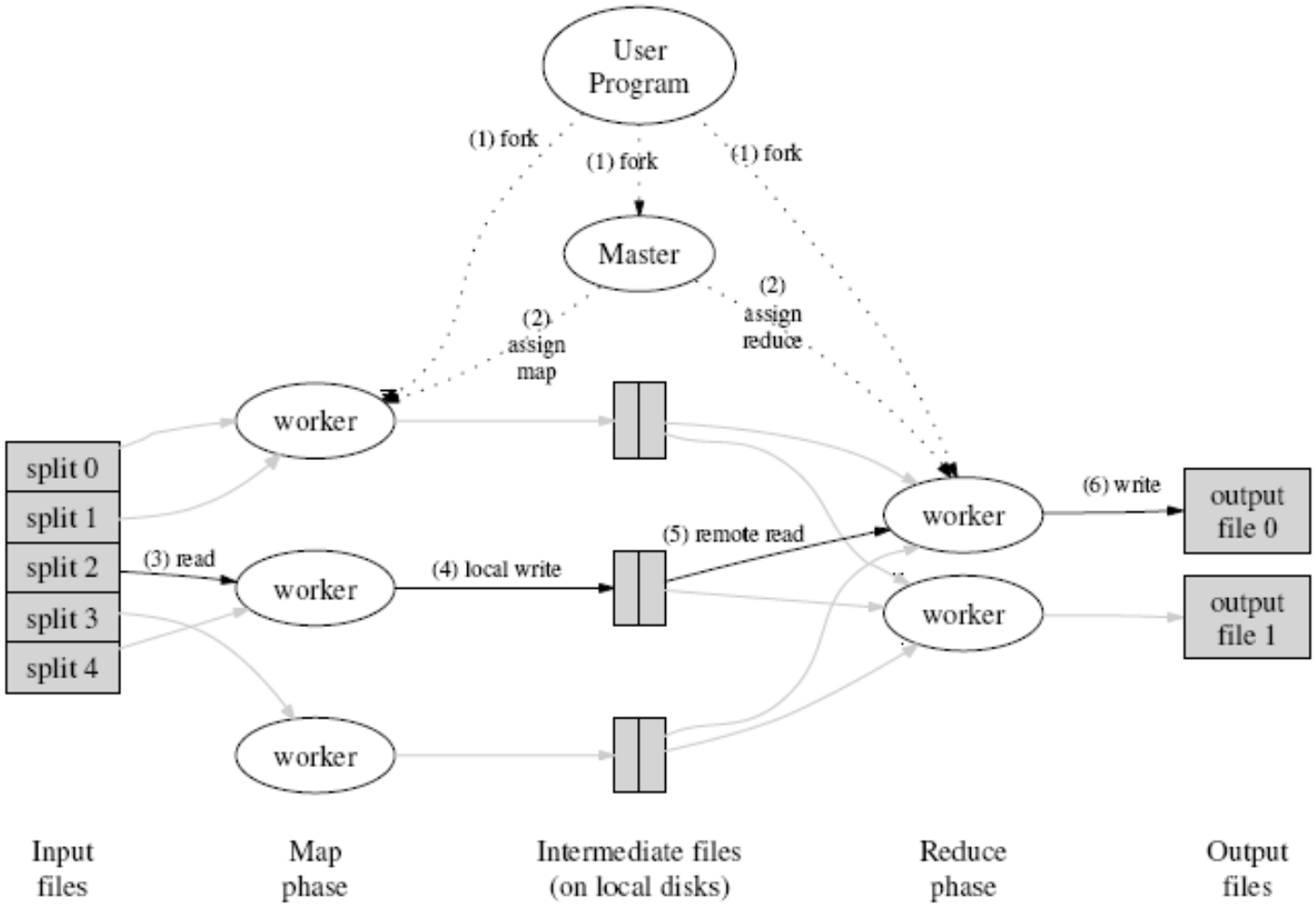


Figure: Source : http ://code.google.com/intl/fr/edu/parallel/mapreduce-tutorial.html

Quid de l'avenir ?

- Cela pose directement la question du traitement des grandes volumétries de données
- Que se soit en physique des particules (LHC), en astronomie (par exemple Sloan Digital Sky Survey : www.sdss.org) ou en biologie, les volumes de données générés sont considérables
  - selon Cisco, 966 exa-octets par an de transfert de données sur le réseau (1 Exa-octet en 2004) (pour mémoire 1 exa-octet : 10^{18} octets et 1 peta-octet : 10^{15} octets)
  - 1,790 milliards à 2,230 milliards de personnes en 2012
  - quelques chiffres généraux ici : `http://netforbeginners.about.com/od/weirdwebculture/f/How-Big-Is-the-Internet.htm` et là `http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf`
  - Les échelles sont désormais l'exa-octet (10^{18}) et le zéta-octet (10^{21})
- Le nouveau défi est de pouvoir traiter ces volumétries
- Par exemple :
  - recherche d'informations
  - recherche de corrélations statistiques entre données

Quid de l'avenir ?

- Les implications et applications sont considérables :
  - traduction automatique entre langues différentes : cf expérimentation Google
  - capacité à suivre statistiquement l'Internet : mouvements d'opinions, thématiques émergentes, analyses stratégiques (recherches en cours, "lieux où l'on pense", centre de gravité économiques, etc.)
- Et la recherche en tant que telle ?
- Les volumétries de données générées par les processus expérimentaux n'ont sans doute pas été exploitées totalement
- Combien de découvertes latentes dans les données actuelles ?
- Va-t-on assister à un nouveau paradigme en sciences, plus centré sur les données que sur les hypothèses scientifiques ou même l'expérimentation ?
- L'avenir le dira ...

En guise de fin

"Autrefois, les physiciens répétaient les expériences de leurs collègues pour se rassurer. Aujourd'hui ils adhèrent à FORTRAN et s'échangent leurs programmes, bugs inclus." E. Dijkstra

Merci de votre attention

Des questions ?