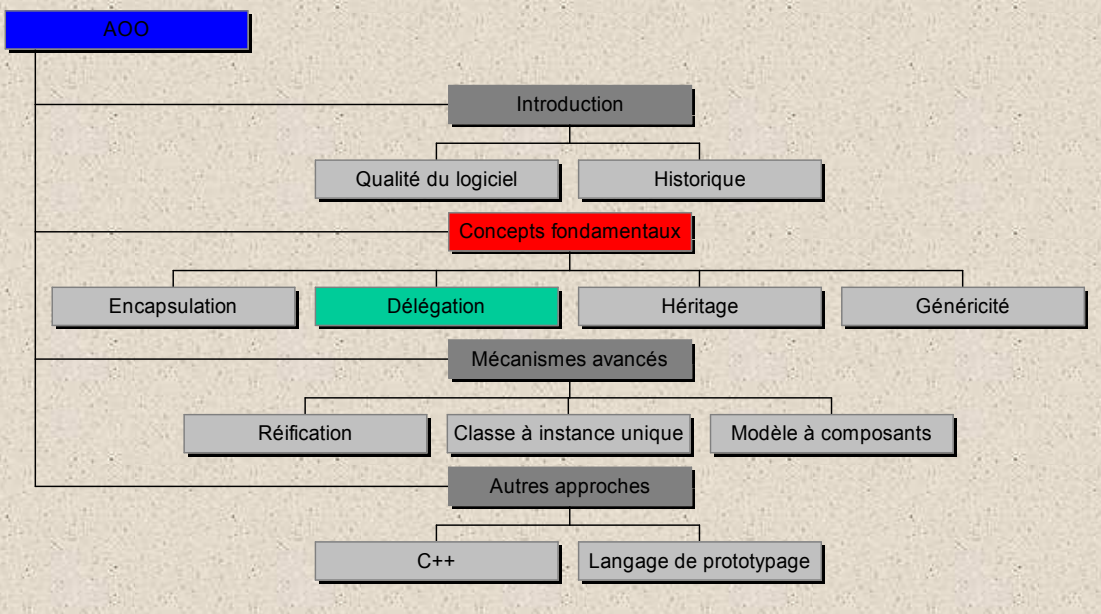
Délégation & Assertion

Programmation par contrat

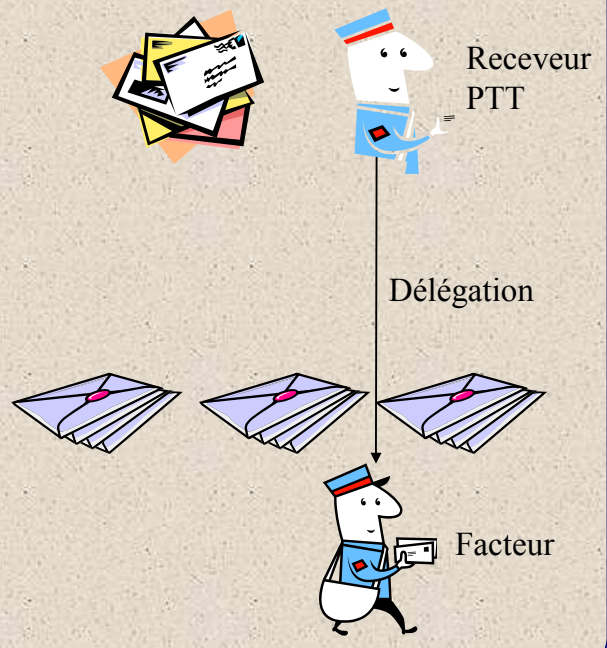
Sommaire



Le concept de Délégation

La notion d'agent

- Un objet est un agent capable de s'exécuter indépendamment de la globalité de son environnement s'il est correctement connecté
- L'activité d'un agent se caractérise par un ensemble d 'états et de transitions
- Exemple: employé d'une entreprise
  - Exécute des tâches demandées par un autre employé
  - Utilise les services d'autres employés pour ce faire
- DELEGATION= utilisation de service suivant un protocole prédéterminé



Délégation : Objet et Classe

Soient Objet1 et Objet2 deux objets d'une application

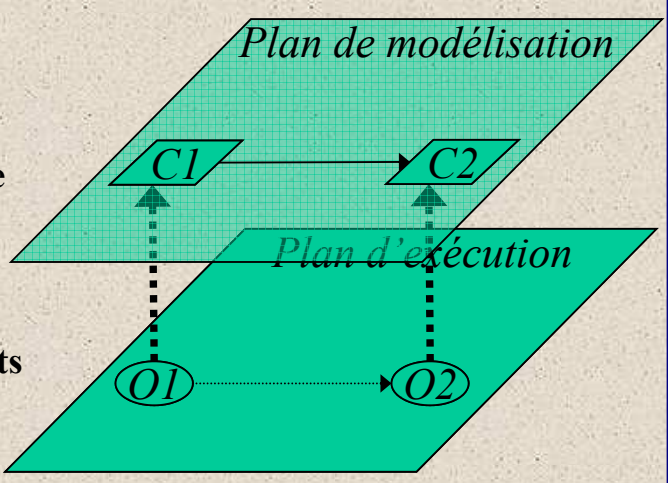
Objet1 est instance de la classe C1

Objet2 est instance de classe C2

Objet1 délégue à Objet2 une partie de son activité

Les classes C1 et C2 sont associées par la relation CLIENTE DE explicitant la délégation entre objets de ces classes.

- C1 est la classe CLIENTE et C2 est la classe SERVEUSE
- La classe cliente va posséder une référence du type de la classe SERVEUSE.

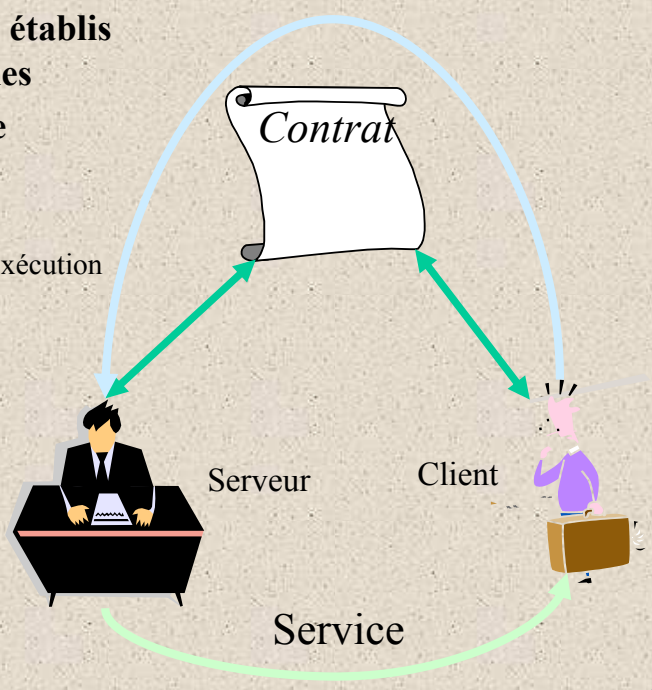


Notion de contrat

Les liens de délégation n 'étant pas établis au départ, il faut fixer des protocoles

Etablissement d'un contrat entre le demandeur et le fournisseur

- Le demandeur du service
  - respecte les prérequis exigés pour l'exécution du service
  - il en attend le service rendu
- Le fournisseur du service
  - assure l'accomplissement du service
  - est assuré des prérequis



Nature du contrat

Le contrat est déterminé par le fournisseur

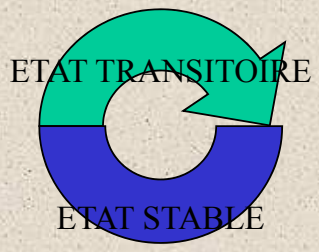
Termes du contrat

- Identification du service : Nom de la méthode
- Nature des informations échangées : Paramètres
- Conditions à satisfaire par le demandeur (prérequis)
  - Conditions sur les valeurs fournies
  - Conditions sur l'état du fournisseur
- Conditions à satisfaire par le fournisseur (service rendu)
  - Conditions sur la valeur retournée
  - Conditions sur l'état du fournisseur

Le Concept d'assertion

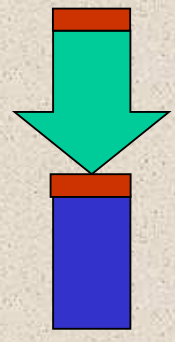
Une assertion est l'expression d'une propriété que doit avoir un objet à des instants précis de son existence.

- au début de l'exécution d'une méthode particulière de l'objet
- à la fin de l'exécution d'une méthode particulière de l'objet
- à l'état stable de l'objet



Ces instants sont déterminés par des points dans le programme

- en tête de méthode :pré-condition (prérequis)
- en fin de méthode : post-condition (service rendu)
- pour l'état stable : invariant



Cohérence d'une classe

Notation

- $PRE_{(X,Y)}$ = pré-condition de la méthode X de la classe Y
- $POST_{(X,Y)}$ = post-condition de la méthode X de la classe Y
- $INV_{(X)}$ = invariant de la classe X
- $DO_{(X,Y)}$ = exécution de la méthode X de la classe Y
- $METH_{(X)}$ = l'ensemble des méthodes de la classe X
- $CSTR_{(X)}$ = l'ensemble des constructeurs de la classe X

Propriétés

- $\forall M \in CSTR_{(C)}$
  - $\{PRE_{(M,C)}\}\ DO_{(M,C)}\ \{POST_{(M,C)} \wedge INV_{(C)}\}$
- $\forall M \in METH_{(C)} - CSTR_{(C)}$
  - $\{PRE_{(M,C)} \wedge INV_{(C)}\}\ DO_{(M,C)}\ \{POST_{(M,C)} \wedge INV_{(C)}\}$

Utilité des assertions

Définition à priori

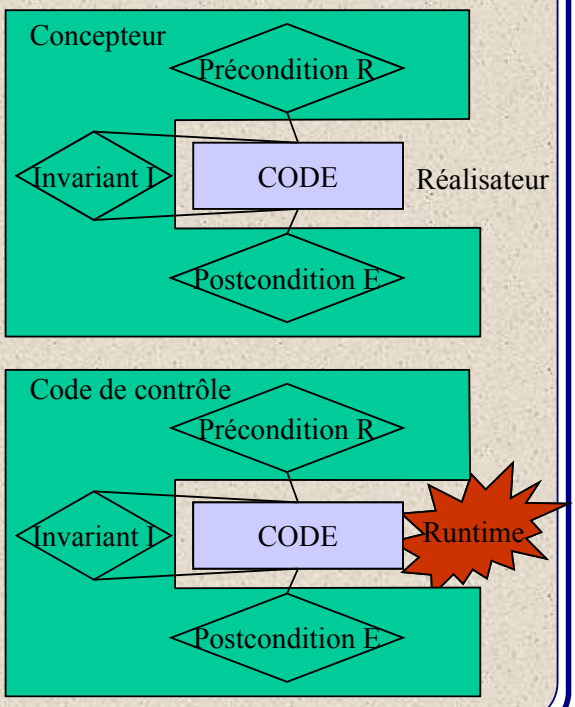
- Spécifier avant de réaliser
  - La pré-condition et la post-condition ainsi que l'invariant de classe sont autant d'éléments qui **spécifient** le code d'une méthode.

Utilisation à posteriori

- Faire un contrôle dynamique des propriétés
  - Le système à l'exécution peut vérifier les propriétés définies dans les assertions et provoquer une exception dans le cas ou une de celles-ci ne serait pas assurée.

Assertion & portée

- Les opérandes utilisés dans une assertion doivent faire partie de l'interface de la classe



INDICE

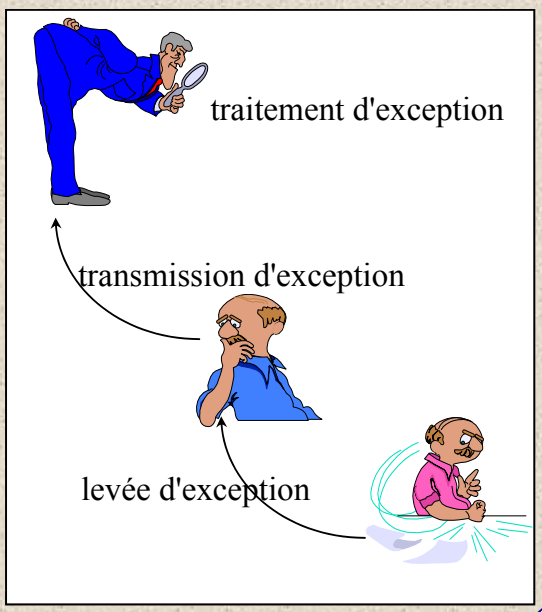
```
/** @invariant coherent : binf() <= valeur() && valeur() <= bsup() */

public class Indice {
   /** création d'un indice dans l'intervalle [binf..bsup]
   * @require valide : binf <bsup
   * @ensure correct : binf()==binf&&bsup()==bsup&&valeur()==binf()
   */
   public Indice(int binf, int bsup) {…}

   /** affecte l'indice avec la valeur de l'indice argument.
   * @require valide : binf()<=i.valeur() && i.valeur() <=bsup()
   * @ensure correct : valeur()==i.valeur()
   */
   public void set(Indice i) {…}
   /** fait progresser l'indice d'une unité s'il n'est pas sur Bsup
   * @ensure correct : valeur()  < bsup() => valeur()==valeur()+1
   * @ensure correct : valeur() == bsup() => valeur()==valeur()
   */
    public void succ() {…}
   /** retourne la valeur de l'indice
   */
    public int get() {…}
   /** accesseurs */
   public int valeur() {return valeur;}
   public int binf() {return binf;}
   public int bsup() {return bsup;}
}
```

Avant exécution

Notion d'exception

Une exception est l'apparition d'une situation anormale

- elle provient de la **violation** d'une **assertion**
  - liée à la logique de l'application
    - IndexOutOfBounds
    - NullPointer
    - Arithmetic
    - ...
  - liée à l'environnement
    - OutOfMemory
    - VirtualMachine
    - ...
- elle est levée dynamiquement
  - Explicitement par du code utilisateur
  - Implicitement par le runtime
- elle n'est jamais ignorée (sauf exception !!!)
  - Prise en compte par un traitement d'exception
  - ré-émise



Les opérateurs des exceptions

On dispose de trois constructions permettant de réaliser les trois traitements des exceptions

- Levée d'une exception (création) instruction throw
- Traitement d'une exception (catch) instruction try
- Transmission d'une exception clause throws

L'instruction throw

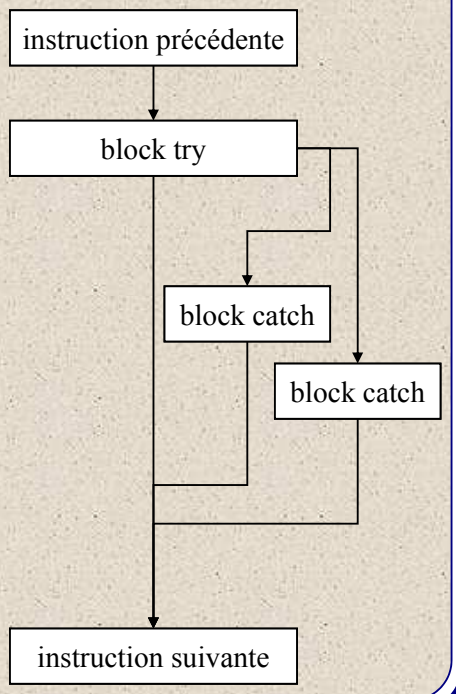
- Elle permet d'engendrer une nouvelle exception
  - Interrompre l'exécution normale
  - Construire un objet exception
    - Contenant les informations décrivant les conditions de la situation anormale.
  - throw e;
    - » e : objet instance de la classe de l'exception
  - throw new <Constructeur de la classe d'exception>
    - Cette forme est la plus courante car l'objet d'exception n'a besoin d'exister que lorsque l'exception intervient.
    - Equivalent à :
      - » { Exception e = new <Constructeur de la classe d'exception>; throw e;}

Les opérateurs des exceptions

l'instruction try-catch

- try {
  <block try>

} catch (< déclaration 1 variable d'exception>) {
  <block catch>
} catch (< déclaration 1 variable d'exception>) {
  <block catch>
}



Les opérateurs des exceptions

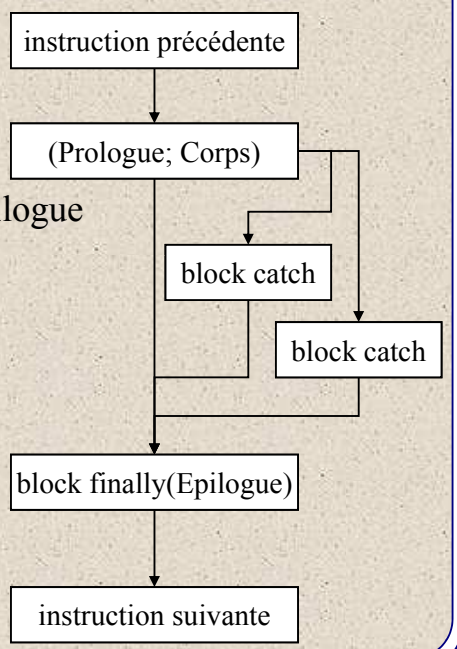
Lorsque le <block try> est non monotone, c-a-d :

– Il est constitué de :
  - Prologue
  - Corps
  - Epilogue

– Qui nécessite l'exécution systématique de l'épilogue

l'instruction try-catch-finally

- try {<block try>
  } catch (< déclaration 1 variable d'exception>) {
  <block>
  } catch (< déclaration 1 variable d'exception>) {
  <block>
  } finally {
  <block>
  }

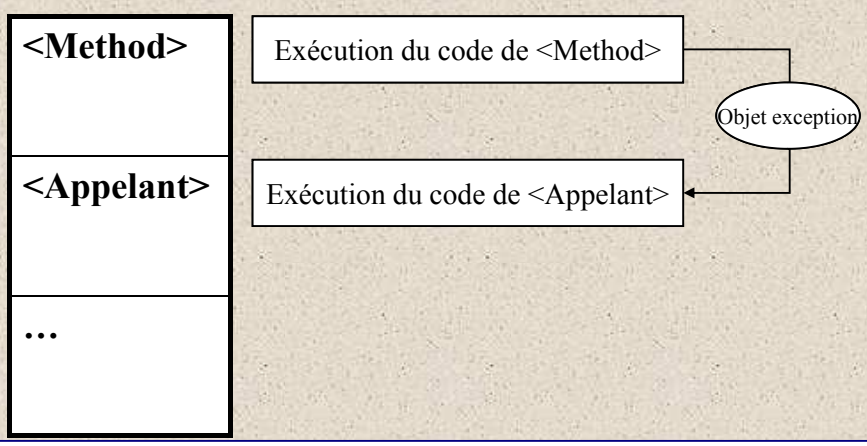


Les opérateurs des exceptions

La clause throws

– Si une exception n'est pas « catchée », elle doit être ré-émise. C'est la partie opératoire de la clause throws.

• <type> <Method>(…) throws <classe d'exception E> {
<block engendrant l'exception E non « catchée »>
}

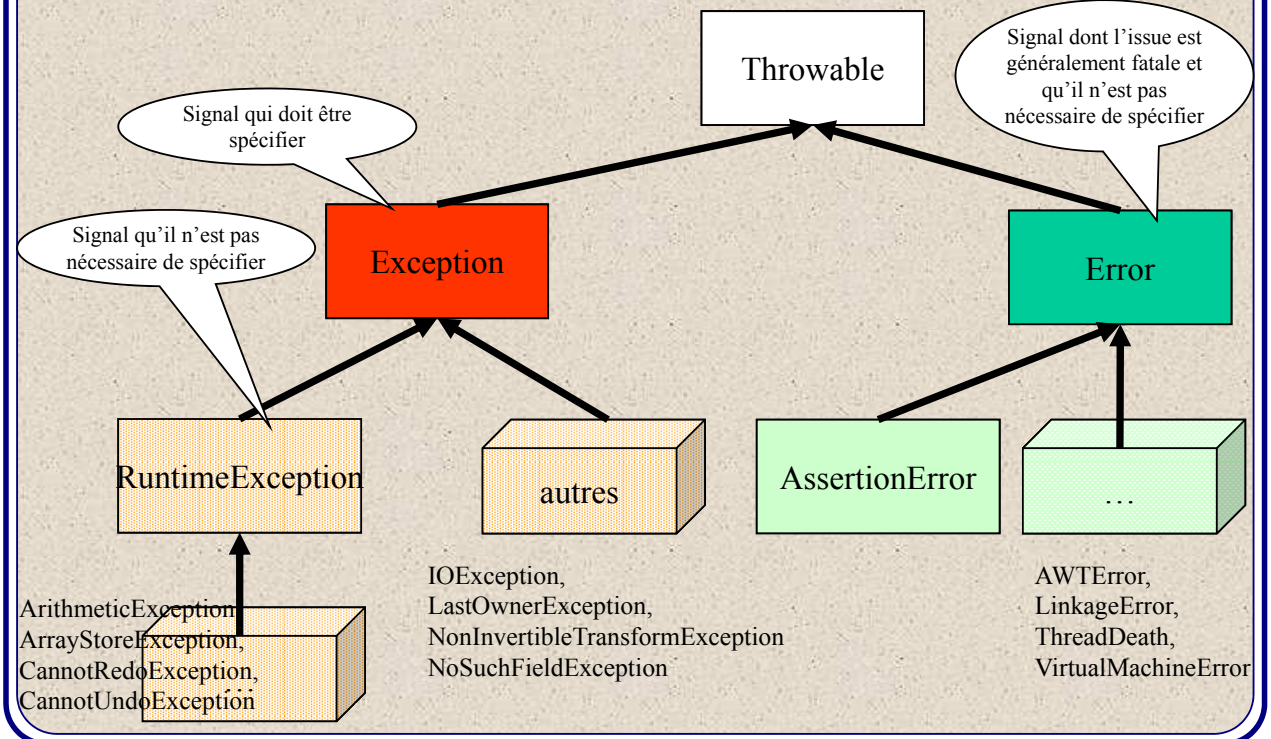


Intérêt du mécanisme d'exception

– Clarification du code
  • par une séparation claire
    – du code de la fonctionnalité
    – des traitements exceptionnels
    – de l'expression des conditions anormales
– Fournir un modèle d'exécution
  • permettant le déroutement
    – contrôlé du flot d'exécution
  • construisant une information d'anomalie.
    – objet exception
    – flot de contrôle
    – rapport d'échec

```
public class C2
public void m2() {
  try{
    C3 obj3;
    …
    obj3.m3();
    …
  catch(MyException e) {
    System.out.println(e);
  }
}
public class C3
public void m3() throws MyException{
  if(Double.isNaN(x))
    throw new MyException();
  …
  y = x / 0.0;
}
```

Hiérarchie des exceptions



Throwable

Constructor Summary

Throwable() Constructs a new throwable with null as its detail message.
Throwable(String message) Constructs a new throwable with the specified detail message.
Throwable(String message, Throwable cause) Constructs a new throwable with the specified detail message and cause.
Throwable(Throwable cause) Constructs a new throwable with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause).

Method Summary

Throwable	**fillInStackTrace**() Fills in the execution stack trace.
Throwable	**getCause**() Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	**getLocalizedMessage**() Creates a localized description of this throwable.
String	**getMessage**() Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace**() Provides programmatic access to the stack trace information printed by printStackTrace().
Throwable	**initCause**(Throwable cause) Initializes the *cause* of this throwable to the specified value.
void	**printStackTrace**() Prints this throwable and its backtrace to the standard error stream.
void	**printStackTrace**(PrintStream s) Prints this throwable and its backtrace to the specified print stream.
void	**printStackTrace**(PrintWriter s) Prints this throwable and its backtrace to the specified print writer.
void	**setStackTrace**(StackTraceElement[] stackTrace) Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
String	**toString**() Returns a short description of this throwable.

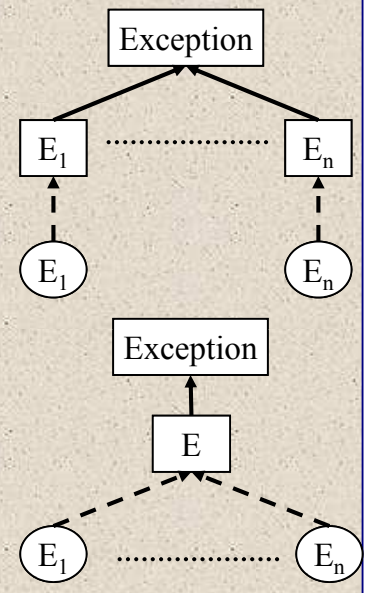
StackTraceElement

Method Summary	
boolean	**equals**(Object obj) Returns true if the specified object is another StackTraceElement instance representing the same execution point as this instance.
String	**getClassName**() Returns the fully qualified name of the class containing the execution point represented by this stack trace element.
String	**getFileName**() Returns the name of the source file containing the execution point represented by this stack trace element.
int	**getLineNumber**() Returns the line number of the source line containing the execution point represented by this stack trace element.
String	**getMethodName**() Returns the name of the method containing the execution point represented by this stack trace element.
int	**hashCode**() Returns a hash code value for this stack trace element.
boolean	**isNativeMethod**() Returns true if the method containing the execution point represented by this stack trace element is a native method.
String	**toString**() Returns a string representation of this stack trace element.

Classifier les exceptions

Pour définir un ensemble {E_1, …, E_n} d'exceptions, on peut :

- Définir les classes d'exception E_1, …, E_n
  - Toutes les instances d'une classe d'exception seront équivalentes,
- Définir une seule classe d'exception E
  - Distinguer les instances par des attributs qui caractérisent chaque «*type*» d'exception {E_1,…,E_n}
- Combiner les 2 solutions précédentes
  - Nommer les exceptions permet de mieux identifier celles-ci
  - Trop de classes d'exception ou des classes d'exception trop proches peut alourdir le système
  - Intérêt de typer les pré-conditions
    - Nul n'est autorisé à **ignorer** ses engagements, c'est la partie spécification de la clause throws



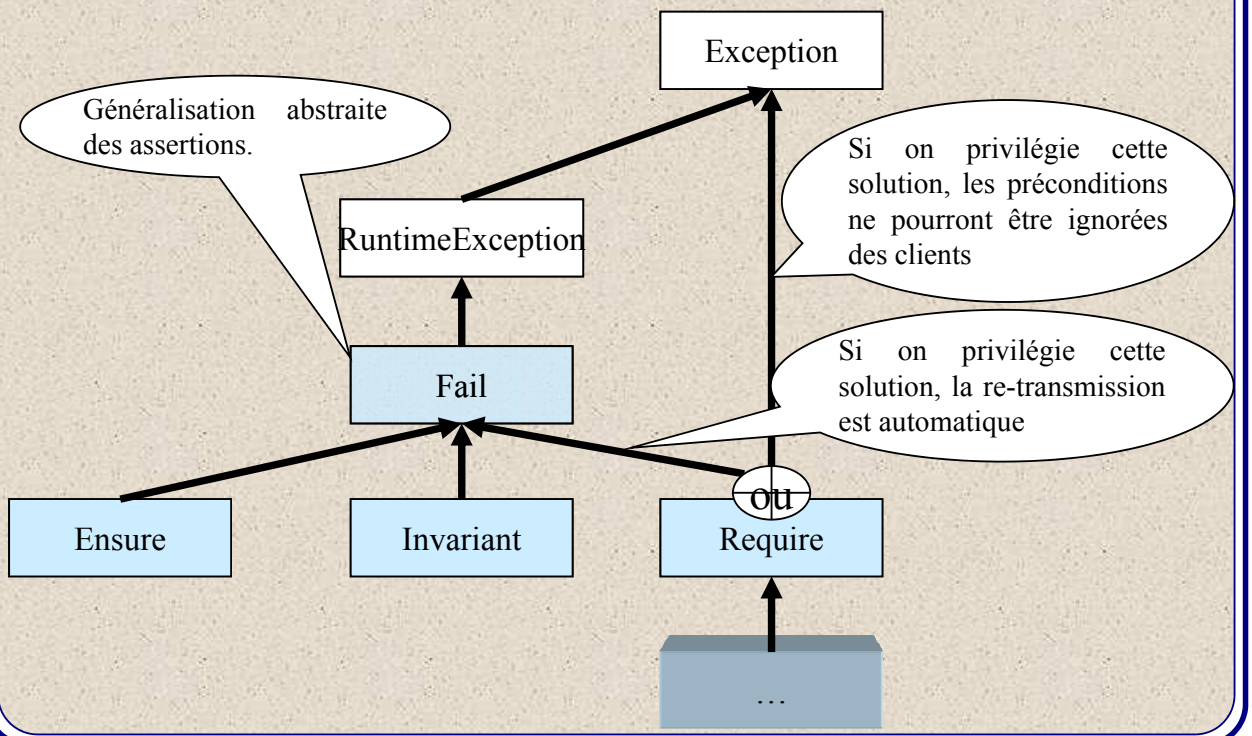
Implantations des assertions

Première solution

– On place les vérifications en début et fin de méthode
- Sous la forme : if(!<condition>) throw new <assertion>(…);
- Le contrôle est systématiquement présent dans le code
- Le contrôle est systématiquement exécuter.
  – Ce peut être pénalisant dans certains cas !

```
class <Class> {
   public <type> <Method>( … ) throws Require {
      if(!<PreCondition>)   throw new Require(…);
      <code effectif>
      if(!<PostCondition>) throw new Ensure(…);
      <Class>_invariant();
   }
   …
   protected void <Class>_invariant() {
      if(!<Invariant>) throw new Invariant(…);
   }
}
```


Hiérarchie des assertions



Les classes d'assertion

Constructor Summary

protected	`Fail(int code, java.lang.String formule)`
protected	`Fail(java.lang.String nom, java.lang.Exception exception)`
protected	`Fail(java.lang.String nom, java.lang.String formule)`

Method Summary

int	`code()` Restitue le code interne de l'exception
java.lang.Exception	`exception()` Restitue l'exception originale
java.lang.String	`formule()` Restitue la condition non vérifiée

Constructor Summary

	`Require(int code, java.lang.String formule)`
	`Require(java.lang.String nom, java.lang.Exception exception)`
	`Require(java.lang.String nom, java.lang.String formule)`

Method Summary

int	`code()` Restitue le code interne de l'exception
java.lang.Exception	`exception()` Restitue l'exception originale
java.lang.String	`formule()` Restitue la condition non vérifiée

Constructor Summary

	`Invariant(int code, java.lang.String formule)`
	`Invariant(java.lang.String nom, java.lang.Exception exception)`
	`Invariant(java.lang.String nom, java.lang.String formule)`

Constructor Summary

	`Ensure(int code, java.lang.String formule)`
	`Ensure(java.lang.String nom, java.lang.Exception exception)`
	`Ensure(java.lang.String nom, java.lang.String formule)`

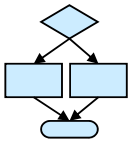
Classe INDICE

```
class Indice {
    protected void Indice_invariant() {
        if(!(binf()<=valeur() && valeur()<=bsup()))
            throw new Invariant(" !(binf()<=valeur() && valeur()<=bsup())");
    }
    public Indice(int bI, int bS)  throws Require {
        if(!(bI>bS)) throw new Require("!(bI< bS)");
        binf=bI; bsup=bS; valeur=binf;
        if(!(binf()==bI)) throw new Ensure("!(binf()==bI)");
        if(!(bsup()==bS)) throw new Ensure("!(bsup()==bS)");
        if(!(binf()==valeur())) throw new Ensure("!(binf()==valeur())");
        Indice_invariant();
    }
    public void succ {
        int _valeur = valeur();
        if(valeur<bsup) valeur++;
        if(!(_valeur==bsup||valeur()==_valeur+1))
            throw new Ensure("!(_valeur() <bsup => valeur()=_valeur()+1)");
        Indice_invariant();
    }
```


Classe INDICE(suite)

```
    public void set(int e) throws Require {
        if(!(binf()<=e  && e<=bsup())) throw new Require("!(binf()<=e<=bsup())");
        valeur=e;
        if(!(valeur()== e)) throw new Ensure("!(valeur()==e)");
        Indice _invariant();
    }
    public int get() {
        int returned = valeur;
        if(!(returned==valeur()) throw new Ensure("!(returned==valeur())");
        Indice _invariant();
        return returned;
    }
    ....
}
```

Nécessite un modèle procédural à sortie unique



Implantations des assertions

Seconde solution

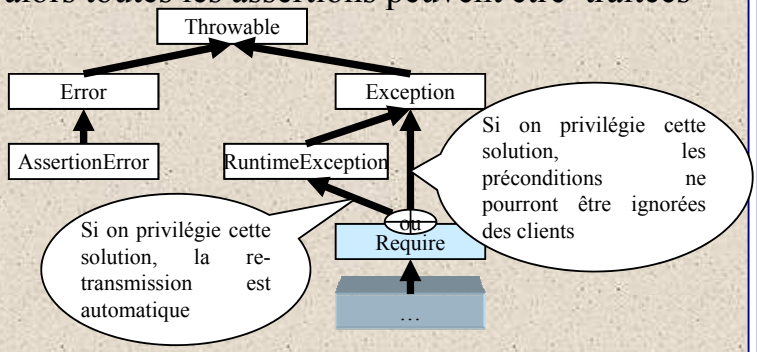
- On réalise ces vérifications avec l'instruction assert (à partir de Jdk1.4)
  - Sous la forme assert(<condition>);
    - Émet l'exception AssertError si la condition n'est pas vérifiée
  - Non conseillé pour les préconditions
    - Pour permettre de différencier les exceptions et donc les causes
    - *Pour obliger la prise en compte de l'exception*
  - Le contrôle est systématiquement dans le code
  - On peut indiquer à la JVM, pour une exécution particulière, si l'on veut ou non exécuter le contrôle
    - On minimise le coût d'exécution
      - » Option de l'interprète : {ea | da | esa | dsa}:{<packagename>|<classname>}*
    - Le code est : if(!$assertionsDisabled && !(<condition>)) throw new AssertionError();
  - On perd le bénéfice des classes d'exception
    - Réduit à AssertionError

Classe INDICE

```
class Indice {
    protected void Indice_invariant() {assert((binf()<=valeur() && valeur()<=bsup()));}
    public Indice(int bI, int bS)  throws Require {
        if(!(bI<bS)) throw new Require("!(bI<bS)");
        binf=bI; bsup=bS; valeur=binf;
        assert((binf()==bI)); assert((bsup()==bS)); assert((binf()==valeur());
        Indice_invariant(); }
    public void succ() {
        int  _valeur = valeur();
        if(valeur<bsup) valeur++;
        assert((_valeur==bsup||valeur()==_valeur+1)); Indice_invariant(); }
    public void set(int e) throws Require {
        if(!(binf()<=e  && e<=bsup())) throw new Require("!(binf()<=e<=bsup())");
        valeur=e;
        assert((valeur()== e)); Indice _invariant(); }
    public int get() {
        int returned = valeur;
        assert((returned==valeur()); Indice _invariant();
        return returned; }
    …
}
```


Hiérarchie des assertions

Si on choisit la solution runtime alors toutes les assertions peuvent être traitées par assert.




```
class Indice {
    public void Indice_invariant() { assert((binf()<=valeur() && valeur()<=bsup())); }
    public Indice( int bI, int bS) { binf=bI; bsup=bS; valeur=binf; }
    public void Indice_int_int_pre( int bI, int bS) {assert(bI<bS)) ;}
    public void Indice_int_int_post( int bI, int bS) { assert((binf()==bI && bsup()==bS && binf()==valeur());}
    public void succ { if(valeur<bsup) valeur++; }
    public void succ_post(int_valeur) {assert((_valeur==bsup()||valeur()==_valeur+1));}
    public void set(int e) { valeur=e; }
    public void set_int_pre(int e) { assert(binf()<=e  && e<=bsup;}
    public void set_int_post(int e) {assert((valeur()== e));}
    public int get() { return valeur; }
    public void get_post(int returned) {assert((returned==valeur());}
    ....
}
```


Implantations des assertions

troisième solution

- La prise en compte des exceptions n'est pas un problème du code émetteur mais du code appelant (mise en œuvre du contrat)
  - Le traitement des assertions doit être placé au point d'appel de la méthode
    - Un appel interne de méthode n'implique pas la vérification des assertions,on différencie les 2 formes :
      - » <M>(…);
      - » this.<M>(…);
  - Les assertions sont implantées comme des méthodes sans effet de bord
    - Chaque méthode peut posséder les méthodes de :
      - » pré-condition
      - » post-condition
    - On associe à la méthode public <T> <M>(<T1>p1, …, <Tn>pn) les méthodes :
      - » public void <M_T1_…_Tn_pre>(<T1>p1, …, <Tn>pn) { …}
      - » public void <M_T1_…_Tn_post>(<T1>p1, …, <Tn>pn) { …}
  - La JVM devrait gérer les appels à ces méthodes
    - Avec un mécanisme permettant d'activer ou d'inhiber la vérification

Classe INDICE : autre réalisation 1

On crée une classe qui contient l'ensemble du code de contrôle.
Ce code est ainsi confiné dans une unité de chargement manipulable globalement (voir chapitre sur l'imbrication).

```
class Indice {
   public Indice( int bI, int bS) { binf=bI; bsup=bS; valeur=binf; }
   public void succ() { if(valeur<bsup) valeur++; }
   public void set(int e) { valeur=e; }
   public int get() { return valeur; }
   ....
   public class Assert {
      public void _invariant() {
            assert((binf()<=valeur() && valeur()<=bsup())); }
      public static void Indice_int_int_Pre(int bI, int bS) throws Require {
            if(!(bI<bS)) throw new Require("!(bI<bS)");}
      public void Indice_int_int_Post(int bI, int bS) {
            assert((binf()==bI)); assert((bsup()==bS)); assert((binf()==valeur()));}
      public void succ_Post(int _valeur) {
            assert((_valeur==bsup()||valeur()==_valeur+1));}
      public void set_int_Pre(int e) throws Require {
            if(!(binf()<=e  && e<=bsup())) throw new Require("!(binf()<=e<=bsup())");}
      public void set_int_Post(int e) {assert((valeur()== e));}
      public void get_Post(int returned) {assert((returned==valeur());
   }
}
```


Classe INDICE : autre réalisation 2

Dans cette version, on dispose de 2 versions de la classe, une sans les vérifications et l'autre avec. On utilise une fabrique spéciale pour créer une ou l'autre version.

```
class Indice {
      public static Indice new(int bI,int bS) {
            if(AssertOn) return new Indice(bI,bS); else return new Indice.Indice(bI,bS);}
      public Indice( int bI,int bS) { binf=bI; bsup=bS; valeur=binf; }
      public void succ() { if(valeur<bsup) valeur++; }
      public int get() { return valeur; }
      ....
      private class Indice extends Indice {
            protected void _invariant() {assert((binf()<=valeur() && valeur()<=bsup()));}
            public Indice ( int bI,int bS) {
                  assert(bI<bS);
                  binf=bI; bsup=bS; valeur=binf;
                  assert((binf()==bI)); assert((bsup()==bS)); assert((binf()==valeur()); _invariant(); }
            public void succ (){
                  int _valeur = valeur(); Indice.succ(); assert((_valeur==bsup||valeur()==_valeur+1)); _invariant(); }
            public int get() {
                  int returned = Indice.get(); assert((returned==valeur()); _invariant(); return returned; }
             ....
}
```