

# Les exceptions

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



*Préservez l'environnement,  
n'imprimez pas ce document*

## TABLE DES MATIERES

1.	Comment utiliser les exceptions ? .....	3
1.1	Rouge .....	4
1.2	Orange .....	5
1.3	Vert.....	5
2.	Quand gérer les exceptions ? .....	5
3.	La rédaction des méthodes avec les exceptions .....	5
3.1	Ne génère pas d'exception.....	5
3.2	Capturer ou générer une exception .....	6
3.2.1	Approche partielle.....	6
3.2.2	Approche globale .....	7
3.2.2.1	Swap.....	8
3.2.2.2	Copie superficiel .....	10
3.2.2.3	Intégration.....	10
3.3	Les exceptions applicatives .....	11
3.4	Difficultés.....	12
3.5	Conclusion .....	12

## Avant-propos

*Ce document explique comment rédiger correctement une méthode pour bénéficier du mécanisme d'exception de java sans perturber l'application.*

Les exceptions ont été intégrées dans Java pour faciliter la gestion des erreurs. Auparavant, il était nécessaire de vérifier chaque retour de méthode pour détecter les erreurs. Par exemple, une méthode désirant copier un fichier doit :

- ouvrir le premier fichier en lecture
- ouvrir le deuxième fichier en écriture
- tant que l'on n'a pas rencontré la fin du 1<sup>er</sup> fichier
- lire les données du premier fichier
- écrire ces données dans le deuxième fichier
- fermer le deuxième fichier
- fermer le premier fichier

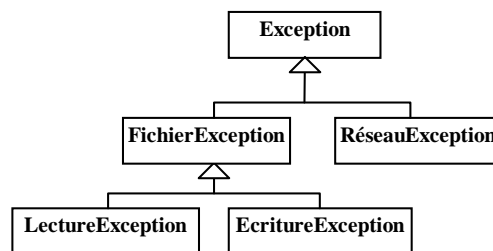
Lors de l'exécution de toutes ces étapes, il peut se produire une foule d'erreurs :

- fichier absent
- ouverture impossible
- lecture erronée
- écriture erronée
- fermeture impossible...

Plutôt que polluer le code par des tests à chaque étape, le développeur va l'encadrer pour qu'il soit géré par une exception. En cas d'erreur, quelle que soit l'étape en cours, elle est interrompue et le code correspondant à sa gestion est exécuté.

- détecte les exceptions
- copie le fichier
- si une erreur arrive pendant le traitement, ferme les deux fichiers.

La capture d'une exception s'effectue sur le *type* de l'objet émis. Par exemple, prenons l'arbre d'héritage suivant, représentant des exceptions :



Des instances de tous ces types peuvent être émises lors d'une exception. Un traitement doit demander la capture d'un objet en indiquant le type attendu. Si un traitement attend une exception de type `FichierException`, il capturera toutes les exceptions de ce type, mais aussi les exceptions des types dérivés. Il capturera des exceptions `FichierException`, `LectureException` et `EcritureException`.

Il faut capturer dans un premier temps les types les plus spécifiques, puis capturer les types de base. Par exemple, la méthode doit effectuer un traitement particulier lors d'une exception `LectureException` et un autre pour les exceptions `FichierException`. Elle doit dans un premier temps capturer une exception `LectureException` et, seulement après, capturer une exception `FichierException`. Sinon, après le traitement de l'exception `FichierException`, il ne lui sera plus possible de capturer l'exception `LectureException`. Elle aura déjà été capturée par `FichierException`. L'ordre de déclaration des exceptions à capturer est important.

## 1. COMMENT UTILISER LES EXCEPTIONS ?

Le code suivant est erroné, pourquoi ?

```
class Planning
{
    Date      debut_;
    Date      fin_;
    Action    action_;

    public void set(Date debut, Date fin)
```

```

    { debut_=new Date(debut.getTime());
      fin_ =new Date(fin.getTime());
    }
}

```

Sans regarder les explications suivantes, essayez de trouver la faille.

La construction d'un objet peut générer une exception `java.lang.OutOfMemoryError`. Dans ce cas, il est possible que l'attribut `debut_` soit modifié sans que `fin_` le soit. Il faut rédiger les méthodes avec une approche transactionnelle. Soit la méthode s'exécute entièrement, soit elle ne s'exécute pas du tout (*commit* ou *rollback*). Les méthodes doivent être « atomiques ». Le code précédent doit être modifié comme ceci :

```

class Planning
{ Date      debut_;
  Date      fin_;
  Action    action_;

  public void set(Date debut,Date fin)
  { Date xdebut;
    Date xfin;

    // Le code suivant peut générer une exception
    xdebut=new Date(debut.getTime());
    xfin  =new Date(fin.getTime());

    // Commit la transaction. Ne peut pas générer d'exception
    debut_=xdebut;
    fin_  =xfin;
  }
  //...
}

```

L'appelant peut capturer une exception `java.lang.OutOfMemoryError` sans perturber le programme.

```

class Application
{ static public void main(String[] args)
  {

    Planning Planning=new Planning();
    Date      debut=new Date();
    Date      fin  =new Date();

    for (;;)
    { try
      { Planning.set(debut,fin);
        //...
        break;
      }
      catch(OutOfMemoryError x)
      { System.err.println("Je n'arrive pas a executer la méthode.\n"+
                          "Libérez de la mémoire !");

        try
        { System.in.read();
          } catch (java.io.IOException e)
          {
          }
        }
      }
    }
}

```

Rédiger un code pour qu'il soit *compatible* avec les exceptions n'est pas facile. Les exceptions peuvent être rares (dérivé de `RuntimeException` ou de `Error`) ou courante (dérivé d'`Exception`). Dans tous les cas, il faut une politique pour les gérer correctement.

Il y a plusieurs approches possibles identifiées par trois niveaux de qualité : vert, orange, rouge. Le niveau vert propose le maximum de qualité.

### 1.1 Rouge

Si une exception arrive, l'instance n'est plus manipulable car elle est dans un état instable. C'est le choix par défaut. La plupart des classes de l'API java utilisent ce niveau pour les exceptions rares. Si une exception arrive, il ne faut plus manipuler les instances. Les exceptions applicatives sont *généralement* gérées correctement.

## 1.2 Orange

Si une exception arrive lors d'une méthode, les instances manipulées sont dans un état stable, pas forcément équivalent à leurs états initiaux ou à la réalisation complète de la méthode. Le traitement n'est pas forcément traité entièrement, mais il ne perturbe pas techniquement l'application.

Les perturbations peuvent être sémantiques. En effet, un utilisateur s'attend à ce qu'une méthode soit entièrement effectuée (ou pas du tout). Orange indique que la méthode est partiellement exécutée.

## 1.3 Vert

Si une exception arrive lors d'une méthode, l'état est restitué comme si la méthode n'avait jamais été appelée (*commit* ou *rollback*). Le traitement est « atomique ». Il n'y a pas de perturbations sémantiques. C'est le niveau maximum de qualité de gestion des exceptions. Dans la pratique, il n'est pas toujours possible de garantir une exception verte. Dans ce cas, il faut se rabattre sur une exception Orange.

Si une méthode manipule d'autres objets répondant au niveau Orange, il est possible qu'elle soit obligée de ne proposer que ce niveau. Le niveau vert exige que toutes les instances manipulées par la méthode ne soient pas modifiées en cas d'exception. Il faut se méfier des effets de bords.

## 2. QUAND GERER LES EXCEPTIONS ?

Il y a trois catégories d'exceptions :

- Les exceptions applicatives (dérivé de `Exception` mais pas de `RuntimeException`) doivent être gérées explicitement car le compilateur impose la capture de l'exception ou sa propagation. Ces exceptions sont également appelées "exception testée" car le compilateur impose leurs gestions.
- Les exceptions techniques (dérivé de `RuntimeException`) ne sont pas signalées par le compilateur. Elles sont considérées comme propagées. Si une exception technique n'est jamais capturée, le programme s'interrompt en affichant une trace de la pile (appel de `printStackTrace()`).
- Les exceptions d'erreurs sont générées par la machine virtuelle (dérivé de `Error`). Elles peuvent ne pas être traitées. Certaines signalent des erreurs de la machine virtuelle, d'autres, l'absence de ressource nécessaire pour continuer l'exécution (`OutOfMemoryError` ou `StackOverflowError`).

Suivant l'application à rédiger, plusieurs approches peuvent être choisies vis-à-vis des exceptions.

- Si l'application est un programme autonome, n'ayant pas une exigence de stabilité importante, ne gérer que les exceptions applicatives est suffisant.
- Si l'application est une librairie à usage interne, vous pouvez ne capturer que les exceptions applicatives. Possédant le source de votre librairie, vous pourrez la modifier, si nécessaire, pour capturer certaines exceptions techniques.
- Si vous développez une librairie ou un framework devant être distribué à grande diffusion, vous ne savez pas les exigences de l'appelant. Vous devez alors gérer les exceptions applicatives et techniques. De plus, vous devez gérer les exceptions concernant les ressources (`OutOfMemoryError` et `StackOverflowError`). Rédiger un framework en tenant compte de cet impératif est très difficile. Vous trouverez ci-dessous des techniques pour faciliter cela.

## 3. LA REDACTION DES METHODES AVEC LES EXCEPTIONS

Pour rédiger une méthode compatible avec les exceptions, il faut bien identifier les risques. Quels sont les traitements pouvant générer une exception ? Quels sont ceux qui n'en génèrent pas ?

Ne génère pas d'exception	Génère des exceptions
<ul style="list-style-type: none"> <li>• La manipulation des types primitifs (sauf division par zéro)</li> <li>• L'affectation d'une référence</li> </ul>	<ul style="list-style-type: none"> <li>• L'appel d'une méthode</li> <li>• L'utilisation d'une variable de classe</li> <li>• La création d'un objet</li> <li>• L'exécution d'une méthode</li> </ul>

### 3.1 Ne génère pas d'exception

Il y a peu de traitement ne générant pas d'exception. En fait, certains peuvent être considérés comme ne générant pas d'exception car les situations d'erreurs sont éliminées lors des tests d'intégrations.

L'appel d'une méthode peut générer une exception (dérivé de `LinkageError`), même si le corps de celle-ci n'en génère pas et que la référence utilisée est différente de `null`. En effet, Java effectue la phase de lien lors de l'exécution. Cette étape peut générer une exception.

Prenons un exemple. Vous utilisez la méthode `a()` de la classe `A` dans une méthode `b()` de la classe `B`. Vous compilez les deux classes. Tous fonctionnent. Ensuite, vous modifiez la classe `A` pour lui supprimer ou pour modifier la signature de la méthode `a()`. Vous compilez la classe `A`, mais pas la classe `B`. Dans cette situation particulière, lorsque la méthode `b()` appelle la méthode `a()`, une exception est générée par la machine virtuelle de Java, car la méthode `a()` n'existe plus.

Ce cas peut être assimilé à une erreur d'intégration et peut être négligé lors de la rédaction des méthodes. Nous pouvons déplacer « L'appel d'une méthode » de la colonne « génère des exceptions » vers la colonne « ne génère pas d'exceptions ».

Pour les mêmes raisons, l'utilisation d'une variable de classe peut générer une exception si celle-ci n'existe plus. Ce cas peut également être négligé car il s'agit d'une erreur d'intégration. Avant la livraison d'un programme java, il faut recompiler toutes les classes.

Nous voyons alors que pour rédiger une méthode ne générant pas d'exception, il faut :

- utiliser les types primitifs,
- faire des manipulations de références,
- appeler des méthodes ne générant pas d'exceptions (attention aux références `null`).

Les méthodes qui ne font que consulter des objets ne génèrent pas d'exceptions (mis à part `LinkageError`). Il est intéressant d'identifier ces méthodes par un commentaire approprié afin de pouvoir rédiger correctement les méthodes atomiques. Un exemple caractéristique de cette démarche est la classe `java.lang.String`. Cette classe ne possède que des méthodes de consultation. Il n'est pas possible de modifier une instance `String`. Toutes les manipulations des `String` sont compatibles avec les exceptions. La construction d'un `String` peut générer une exception, mais une fois l'instance présente, il n'y a plus de problème. L'instance est immuable.

Il est intéressant de séparer les méthodes de consultation des méthodes de modifications. Il existe des patterns permettant de vérifier cela lors de la compilation (Je vais écrire un papier là-dessus). Les méthodes de consultation ne génèrent pas d'exception car elles ne modifient rien.

Maintenant que nous avons identifié ces situations confortables, il faut regarder comment respecter un des niveaux de protection vis-à-vis des exceptions.

### 3.2 Capturer ou générer une exception

Pour respecter le niveau vert, il ne faut pas manipuler les instances directement tant que l'on n'est pas sûr que l'on pourra terminer le traitement. Un traitement atomique est décomposé en trois étapes :

- Une phase d'initialisation prépare le traitement à une approche transactionnelle ;
- le traitement proprement dit ;
- une phase de validation du traitement. Cette phase fige les modifications du traitement.

Les deux premières phases peuvent générer des exceptions. La dernière phase ne doit pas en générer. Pour respecter ce protocole, il y a deux approches possibles.

- L'approche partielle mémorise une partie de l'instance avant d'effectuer le traitement de la méthode.
- L'approche globale travaille sur une copie complète de l'instance.

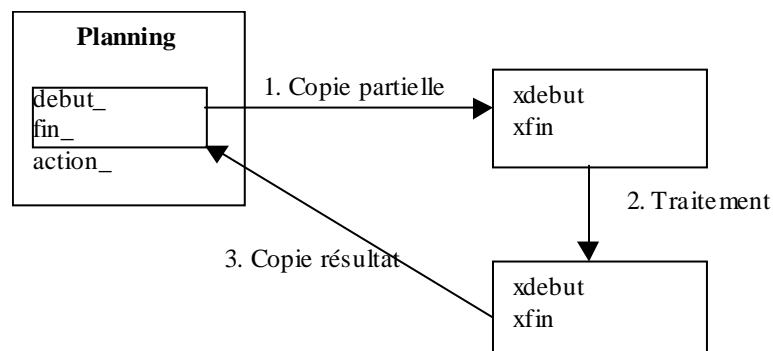
#### 3.2.1 Approche partielle

Il faut dans un premier temps identifier tous les attributs devant être impacté par la méthode ; Déclarez une variable locale pour chacun des attributs, et effectuez le traitement sur ces variables locales. Ensuite, si tout s'est correctement passé, il faut modifier l'instance courante en n'utilisant que des traitements ne générant pas d'exceptions. C'est l'approche utilisée dans le premier exemple.

```
class Planning
{ //...
  public void set(Date debut,Date fin)
  { // Etape 1: Déclaration des variables locales
    //   représentant les attributs manipulés.
    Date xdebut;
    Date xfin;

    // Etape 2: Exécution de la méthode sur les variables locales.
    //   Cette étape peut générer une exception
    xdebut=new Date(debut.getTime());
    xfin  =new Date(fin.getTime());

    // Etape 3: Modification des attributs de l'instance
    //   Ne peut pas générer d'exception.
    debut_=xdebut;
    fin_=xfin;
  }
}
```



Si une instance pointée par un attribut doit également être modifiée par une méthode, il faut la dupliquer vers une variable locale avant d'y faire des manipulations.

```
class Planning
{ //...
  void setAction(Action action)
  { // Etape 1
    Action xaction=new Action(action_); // ou action_.clone();

    // Etape 2
    xaction.set(action); // Exception possible

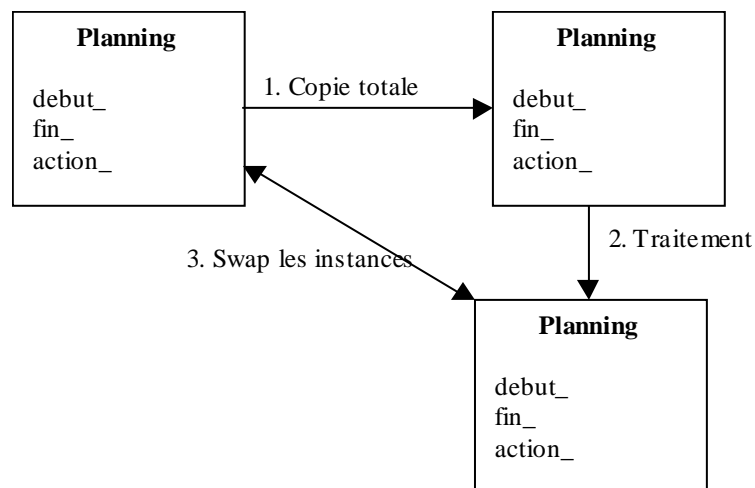
    // Etape 3
    action_=xaction;
  }
}
```

Cette approche est partielle, car on ne copie que les attributs étant directement impacté par la méthode. Attention, il faut faire très attention à l'ensemble des attributs manipulés par la méthode, effet de bord compris. Si la méthode appelle une autre méthode sur l'instance, et que cette autre méthode modifie des attributs, on ne propose que des exceptions Orange. L'objet n'a pas exécuté entièrement la méthode, mais il est dans un état stable. Dans ce cas, il est préférable d'utiliser l'approche globale pour offrir une exception verte.

### 3.2.2 Approche globale

Une autre approche, moins fine que la précédente, permet de faciliter la rédaction des méthodes de modification et permet un niveau vert.

Avant de manipuler l'instance courante, on effectue une copie complète de l'instance. On travaille ensuite sur cette copie, puis on inverse l'état de la copie avec l'état de l'instance courante. Cette inversion ne doit pas générer d'exception.



Modifions la classe `Planning` pour que la méthode `set()` permette également de modifier l'`Action`. La méthode `set()` appelle la méthode `setAction()`.

```
class Planning
{ Date debut_;
  Date fin_;
  Action action_;
  //...
  public void set(Date debut, Date fin, Action act)
  {
    setAction(act);
    debut_=new Date(debut.getTime());
    fin_=new Date(fin.getTime());
  }
  public void setAction(Action act)
  { //...
  }
}
```

Il faut que la méthode `set()` s'effectue entièrement ou pas du tout (qu'elle soit « atomique »). Même si la méthode `setAction()` est de niveau vert, l'instance courante est modifiée après l'appel de `setAction()`, mais peut générer une exception lors de l'appel de `new Date(debut)` ou de `new Date(fin)`. Nous désirons que `set()` ne modifie pas l'instance courante si une exception est générée. Nous devons alors travailler sur une copie de l'instance courante, y exécuter les traitements, puis inverser les attributs de la copie avec les attributs courants.

Il y a deux approches pour obtenir une copie d'un objet.

- l'appel d'un constructeur ;

- ou l'appel de la méthode `clone()`.

La première approche permet de construire une nouvelle instance en utilisant ce que l'on appelle « un constructeur de copie ». Un constructeur de copie est un constructeur recevant un paramètre du même type que la classe et ayant pour fonction de construire une copie du paramètre.

```
class Planning
{ //...
// Constructeur de copie
  Planning(Planning x)
  { debut_=new Date(x.debut_.getTime());
    fin_=new Date(x.fin_.getTime());
    action_=new Action(x.action_);
  }
}
```

Cette approche présente un inconvénient. L'appelant doit connaître le type de l'objet à construire. Si une classe `MonPlanning` hérite de `Planning`, il faut rédiger un constructeur de copie pour la classe `MonPlanning`. L'appelant doit savoir qu'il manipule une instance `Planning` ou `MonPlanning` pour en obtenir une copie.

```
void f(Planning x)
{ Planning copie=new Planning(x);
// ou copie=new MonPlanning(x) ?
}
```

L'approche « constructeur de copie » ne permet pas d'utiliser le polymorphisme. L'approche `clone()` permet de remédier à cet inconvénient.

Nous déclarons la méthode `clone()` dans la classe `Planning` pour générer une nouvelle instance possédant les mêmes valeurs que l'instance courante.

```
class Planning
{ //...
// Constructeur de copie
  Planning(Planning x)
  { ... }
// Création d'une copie de this
  public Object clone()
  { return new Planning(this);
  }
}
```

Nous pouvons alors utiliser cette méthode `clone()` pour travailler sur une copie.

Ensuite, possédant une copie de l'instance courante, le traitement est effectué sur la copie. Une fois le corps de la méthode terminée, il faut recopier l'état de la copie sur l'instance courante. Il y a encore plusieurs approches pour cela.

### 3.2.2.1 Swap

L'approche `swap` inverse les attributs de l'instance courante avec les attributs de la copie.

```
class Planning implements Cloneable
{ //...
// Exemple de methode Atomique
  public void set(Date debut,Date fin,Action act)
  { // Etape 1
    Planning rc=(Planning)clone(); // Generation d'une copie

    // Etape 2
    rc.setAction(act); // Travail sur la copie
    rc.debut_=new Date(debut.getTime());
    rc.fin_ =new Date(fin.getTime());

    // Etape 3
    swap(rc); // Commit la méthode.
  }

  // Swap les attributs de this et de x.
  // Cette methode garantie qu'elle ne génère jamais d'exceptions.
  // Elle ne fait que des manipulations de pointeurs.
  private void swap(Planning x)
  {
    // Swap debut_
    { Date z=debut_;
      debut_=x.debut_;
      x.debut_=z;
    }
    // Swap fin_
    { Date z=fin_;
      fin_=x.fin_;
    }
  }
}
```

```

        x.fin_=z;
    }
    // Swap action_
    { Action z=action_;
      action_=x.action_;
      x.action_=z;
    }
}

```

Ce protocole permet de garantir que `set()` sera exécuté totalement ou pas du tout (Vert).

`setAction()` doit également garantir une exécution atomique. Elle peut utiliser le même modèle et travailler sur une copie.

```

class Planning implements Cloneable
{ //...
  public void setAction(Action act)
  { Planning rc=(Planning)clone();
    //...
    swap(rc);
  }
}

```

En fait, pour exécuter correctement `set()` il y aura deux copies de générées :

- une pour `set()`
- et une autre pour `setAction()`.

Ce n'est pas très efficace. Pour améliorer les performances, il faut proposer pour chaque méthode :

- une version atomique `public`
- et une version non atomique `private`.

Nous obtenons au final la classe suivante :

```

class Planning implements Cloneable
{ Date    debut_;
  Date    fin_;
  Action  action_;

  // Constructeur de copie
  Planning(Planning x)
  { //...
  }

  // Duplication de this
  public Object clone()
  { return new Planning(this);
  }

  // Swap les attributs de this et de x.
  // Cette methode garantie quelle ne génère jamais d'exceptions
  private void swap(Planning x)
  { //...
  }

  // Exemple de methode Atomique
  public void set(Date debut,Date fin,Action act)
  { Planning rc=(Planning)clone(); // Generation d'une copie
    rc.set_(debut,fin,act);       // Travail sur la copie
    swap(rc);                     // Commit la méthode.
  }

  // Version non Atomique de la méthode
  private void set_(Date debut,Date fin,Action act)
  { setAction_(act);              // Travail sur la copie
    debut_=new Date(debut.getTime());
    fin_  =new Date(fin.getTime());
  }

  // Methode Atomique
  public void setAction(Action act)
  { Planning rc=(Planning)clone();
    rc.setAction_(act);
    swap(rc);
  }

  // Version non Atomique

```

```
private void setAction_(Action act)
{ //...
}
}
```

On constate, que les versions non atomiques sont similaires à la rédaction des méthodes s'il n'y avait jamais d'exception en Java. Cela entraîne un confort lors de l'utilisation de l'approche globale. Vous pouvez dans un premier temps rédiger votre méthode en faisant abstraction des exceptions, puis vous intégrerez l'approche atomique par la suite.

### 3.2.2.2 Copie superficielle

Une autre approche utilise une simple affectation des attributs pour terminer la méthode à la place d'un `swap` des attributs.

```
class Planning implements Cloneable
{ //...
  // Copie les attributs de x sur this.
  // Cette methode garantie qu'elle ne génère jamais d'exceptions.
  // Elle ne fait que des manipulations de pointeurs.
  private void shallowCopy(Planning x)
  { debut_=x.debut_;
    fin_=x.fin_;
    action_=x.action_;
  }
}
```

La méthode `shallowCopy` est utilisée à la place de la méthode `swap`. Elle fait une copie des références, et non des objets référencés. L'instance courante et la copie partagent les objets référencés. Cela peut poser des problèmes dans deux situations :

- Si on réutilise la copie,
- ou s'il existe une méthode `finalize()`.

La copie peut être recyclée pour éviter la construction de nouvelles instances lors de l'appel d'une méthode (Cf. « les optimisations avec java »). Dans ce cas, il faut que la copie soit également dans un état stable. `swap` est préférable.

Si la classe possède une méthode `finalize()`, celle-ci sera appelée sur la copie. Dans ce cas, l'originale peut être impacté. Par exemple, si `finalize()` modifie la date de début, l'originale aura une date modifiée lors de la destruction de la copie. L'approche `swap` est préférable à l'approche `shallowCopy`. Pour éviter les effets de bords suite à un appel à `finalize()` il faut éventuellement modifier les attributs de la copie.

```
class Planning implements Cloneable
{ //...
  // Copie les attributs de x sur this.
  // Cette methode garantie qu'elle ne génère jamais d'exceptions.
  // Elle ne fait que des manipulations de pointeurs.
  private void shallowCopy(Planning x)
  { debut_=x.debut_;
    fin_=x.fin_;
    action_=x.action_;
    // Clear copy
    x.debut_=null;
    x.fin_=null;
    x.action_=null;
  }
}
```

La méthode `finalize()` doit tenir compte de cela. Pour des raisons d'optimisation minime, `shallowCopy` peut être utilisé à la place `swap()`.

### 3.2.2.3 Intégration

L'approche globale permet d'offrir un niveau vert si une méthode appelle d'autres méthodes et facilite la prise en compte tardive des exceptions.

Les versions non atomiques sont `private` pour bénéficier des optimisations du compilateur. Elles peuvent être `protected` pour permettre la rédaction de méthode non atomique dans les classes dérivées, ou `public` pour permettre l'appel d'un traitement non atomique faisant intervenir plusieurs instances. Par exemple :

```
class Compte implements Cloneable
{
  // Transfert de fond
  void ecriture(Compte dest,double valeur)
  { // Generation de copies
    Compte csrc=(Compte)clone();
    Compte cdest=(Compte)dest.clone();

    // Travail sur les copies. Peut générer des exceptions
    csrc.credit_(valeur);
    cdest.debit_(valeur);

    // Commit la transaction. Ne peut pas générer d'exception
  }
}
```

```

        swap(csrc);
        dest.swap(cdest);
    }
}

```

Les méthodes `credit_()` et `debit_()` ne sont pas atomiques. Elles sont déclarées `public`. Comme la méthode `ecriture()` utilise des copies locales, ce n'est pas gênant. Cela augmente les performances de la méthode.

### 3.3 Les exceptions applicatives

Nous venons de voir comment rédiger des méthodes avec le maximum d'exigence vis-à-vis des exceptions (Toutes les exceptions sont gérées). Dans la pratique, les applications java n'offrent pas autant de qualités. Seules les exceptions applicatives sont gérées car le compilateur l'impose.

Une première approche consiste à s'inspirer de la classe `java.lang.String` de java. Cette classe propose uniquement des méthodes de consultation. Une méthode de cette classe peut générer une exception mais l'instance courante n'est pas impactée (Vert). Toutes les méthodes de `String` sont atomiques. Vous pouvez rédiger des classes offrant uniquement des accès en lecture. Une fois l'instance construite, il n'est plus possible de la modifier. Bien évidemment, il n'est pas possible de n'offrir que des méthodes de consultations pour une application. Certaines classes sont de bon candidat pour cette démarche, d'autres non.

La rédaction d'un code gérant les exceptions applicatives n'est pas tellement différente de la rédaction d'un code *pure* pour toutes les exceptions. En effet, les instances doivent rester dans un état stable si une exception applicative est détectée. Les méthodes doivent respecter le niveau vert (à défaut, le niveau Orange).

Lorsque le programme capture une exception (`catch`), il doit *faire le ménage* pour restituer les objets dans un état stable. Si la méthode n'impacte que des variables locales, il n'y a pas de traitement particulier à faire. Le ramasse-miettes fera le ménage à votre place. Si elle impacte des instances, il n'est pas toujours évident de capturer une exception car les informations initiales sont généralement perdues. Par exemple, pour la méthode `read` suivante :

```

class Planning
{
    //...
    public void read(DataInput in)
    {
        try
        {
            debut_=new Date(in.readUTF());
            fin_ =new Date(in.readUTF());
        } catch (IOException e)
        { // ???
        }
    }
}

```

Il n'est pas évident de rédiger le traitement à exécuter lors du `catch`. En effet, si une `IOException` arrive lors du changement de l'attribut `fin_`, l'attribut `debut_` est déjà modifié. L'instance `Planning` est dans un état instable.

Il faut utiliser l'approche partielle ou globale par garantir un niveau vert.

```

class Planning implements Cloneable
{
    //...
    // Version atomique
    public void read(DataInput in)
    { Planning rc=(Planning)clone();
      try
      { rc.read_(in);
        swap(rc); // n'est pas executé s'il y a une exception
      } catch (IOException e)
      { //...
      }
    }
    private void read_(DataInput in) throws IOException
    { debut_=new Date(in.readUTF());
      fin_ =new Date(in.readUTF());
    }
}

```

La méthode `read()` peut propager l'exception vers l'appelant (`throws`).

```

class Planning implements Cloneable
{
    //...
    // Version atomique
    public void read(DataInput in) throws IOException
    { Planning rc=(Planning)clone();
      rc.read_(in);
      swap(rc); // n'est pas executé s'il y a une exception
    }
}

```

```
//...
}
```

L'instance courante reste dans un état stable de niveau vert. La méthode `read()` est exécutée entièrement ou pas du tout. Elle est atomique.

Il n'est pas suffisant de se contenter de propager l'exception. Cela entraîne souvent un niveau Rouge (l'objet est instable), et parfois un niveau Orange (l'objet est stable, mais le traitement n'est pas complet). Toutes les gestions des exceptions applicatives doivent être rédigées avec rigueur. Le compilateur signale les exceptions pouvant arriver. C'est au développeur de rédiger correctement le traitement à effectuer. Propager une exception sans appel à `swap` est un signe de qualité insuffisante. Une relecture critique d'un code java peut vous permettre d'identifier ces erreurs.

Un code capturant une exception doit assumer que l'application peut continuer après la capture. À défaut, il est préférable de propager l'exception par un `throws` approprié dans la signature de la méthode. Cette approche décharge le rédacteur de la méthode de la prise en compte de l'exception, avec éventuellement, le choix d'un niveau rouge. L'appelant devra alors faire le nécessaire pour ne plus utiliser l'instance et/ou les instances en relation avec l'instance ayant exécuté la méthode. Au pire, l'exception est capturée dans la méthode `main()` de l'application et interrompt le programme après avoir affiché un message d'excuse. Capturer une exception sans faire correctement le ménage rend très difficile le déverminage des programmes. Il est préférable de ne pas capturer l'exception que de cacher une erreur exceptionnelle par une mauvaise approche de la gestion de l'exception.

### 3.4 Difficultés

Il n'est pas toujours facile de travailler sur des copies d'objets. Par exemple, si le modèle de donnée impose d'avoir deux instances qui se référencent mutuellement (A pointe vers B et B pointe vers A), il est difficile de rédiger la méthode `swap` (ou la méthode `shallowcopy`). En effet, il faut également inverser les pointeurs réciproques.

```
class A
{ B fille;
}

class B
{ A pere;
}
```

La méthode `swap` de la classe B doit être rédigée ainsi :

```
class B
{ A pere;
  private void swap(B x)
  {
    {
      A xpere=pere;
      pere=x.pere;
      x.pere=xpere;
    }
    // Swap des références fille !
    B xfille=pere.fille;
    pere.fille=x.pere.fille;
    x.pere.fille=xfille;
  }
}
```

Nous sommes dans un cas idéal où la méthode `swap` a un accès à l'attribut `fille`. L'attribut n'étant pas qualifié, il possède un accès `package`. A et B appartiennent au même `package`.

Il n'est pas toujours évident de connaître toutes les instances référençant l'objet. Pour respecter le principe d'encapsulation, on ne devrait pas connaître le contenu d'un objet. L'exemple ci-dessus montre que ce n'est pas toujours possible. La classe B doit connaître les attributs de la classe A.

Il faudrait posséder une méthode `become()`, équivalente à celle que propose Smalltalk pour pouvoir transformer une instance en une autre. Automatiquement, tous les pointeurs sur l'originale pointeraient sur la copie. La machine virtuelle de java peut *théoriquement* faire cela (elle possède les informations nécessaires). Avec un outil comme cela, la méthode `swap()` serait remplacée par un appel à `become()` pour inverser l'instance courante et la copie.

### 3.5 Conclusion

Les codes de toutes les méthodes non `private` doivent être atomiques, ou un commentaire doit indiquer la couleur que garantit la méthode, et les types d'exceptions gérées (techniques ou applicative). Les commentaires peuvent, par exemple, suivre ce modèle.

```
/**
 * @exception      IOException [Vert]
 * @exception      Error      [Rouge]
 */
```

`Error` traduit les exceptions techniques.

Voici une démarche pour sélectionner l'approche à utiliser pour une méthode verte.

Est-ce une méthode de consultation ?

Si oui, pas de traitement particulier

Sinon,

La méthode appelle une autre méthode ?

Si oui, utiliser l'approche globale

a. la classe est-elle polymorphique ?

Si oui, utiliser l'approche `clone()`

Sinon, utiliser l'approche « constructeur de copie »

b. effectuer le traitement sur la copie

c. Y a-t-il un risque de réutiliser la copie ou une méthode `finalize()` ?

Si oui, utiliser `swap()`

Sinon, utiliser `shallowcopy()`

Sinon, utiliser l'approche partielle

a. copier les attributs utilisés

b. effectuer le traitement

c. modifier les attributs de l'instance

Pour simplifier, il est préférable de sélectionner toujours l'approche globale avec l'utilisation des méthodes `clone()` et `swap()`. Ce modèle est compatible avec l'ensemble des cas possibles. Choisir une autre approche peut compliquer la modification ultérieure d'une méthode. En effet, le code actuel d'une méthode peut entraîner le choix de l'approche partielle, mais une évolution de la méthode imposera l'approche globale. Pour éviter ces inconvénients, il est préférable d'utiliser la démarche la plus générique, au détriment de la performance. L'approche globale répond correctement à tous les cas.

Nous avons vu que la rédaction des méthodes Java n'est pas aussi simple qu'il y paraît. Une des difficultés est que les APIs Java ne sont pas documentées vis-à-vis des exceptions. On ne sait pas si une instance de l'API reste stable en cas d'exception. Les méthodes de l'API sont *généralement* Rouges pour les exceptions techniques et Orange pour les exceptions applicatives. Alors que Java impose les exceptions, il est difficile de rédiger correctement une méthode atomique. Cette difficulté est généralement cachée dans la littérature.

Rédigez vos méthodes avec des exceptions vertes, à défaut, avec des exceptions orange.

**INDEX**

---

**B**

become()..... 16

---

**C**

catch ..... 14

---

**E**

Error ..... 6

exception

approche globale..... 8

shallow copy ..... 12

exception

applicative..... 5, 14

approche partielle..... 7

Orange ..... 5

Rouge..... 5

technique..... 5

Verte ..... 5

---

**F**

finalize()..... 12

---

**J**

java.lang.String ..... 7, 14

---

**O**

OutOfMemory ..... 6

---

**P**

printStackTrace() ..... 5

---

**R**

RuntimeException ..... 5

---

**T**

throws ..... 15