Module Programmation Objet

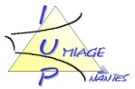
Chapitre 7 - Exceptions et Assertions

Pascal André, Gilles Ardourel

Université de Nantes

2010

Projet DVD Miage



Exceptions

Définition

Definition

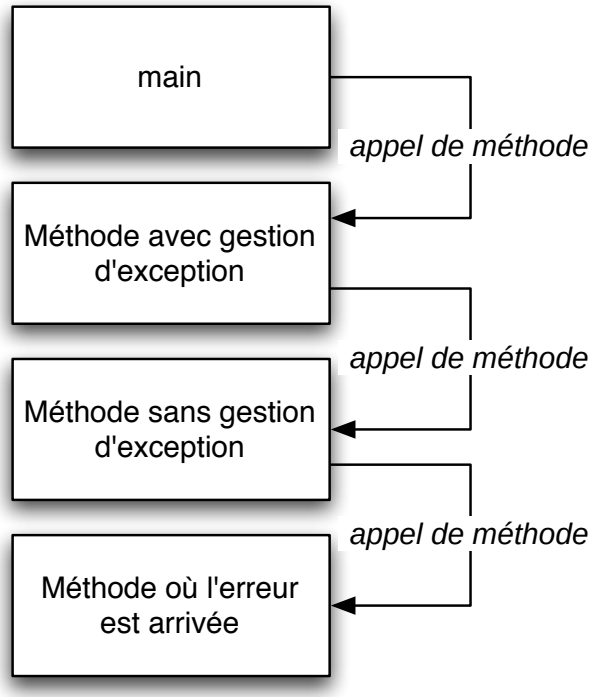
Une *exception* est un événement provoquant l'interruption du flot normal d'exécution d'un programme.

Les exceptions surviennent à la suite d'*erreurs* ou de circonstances *exceptionnelles* justifiant l'arrêt de la procédure en cours.

Les principales opérations les concernant sont leur *émission* et leur *capture*.

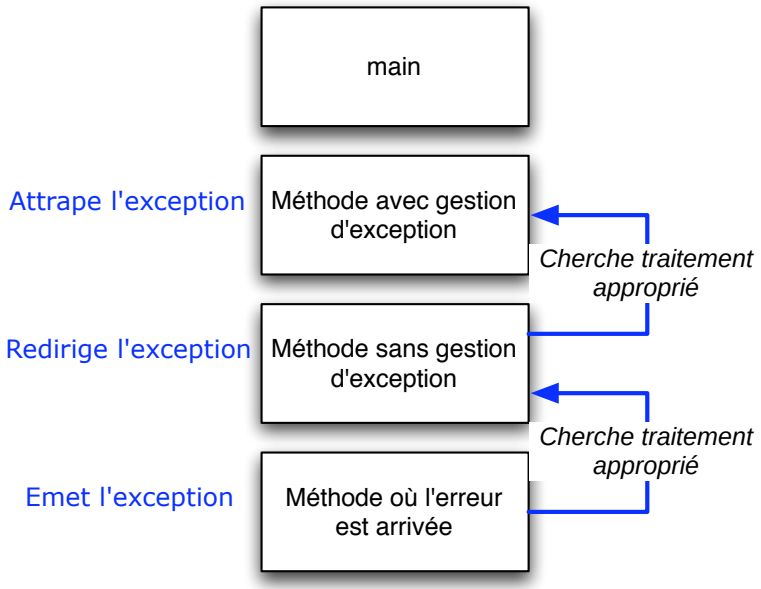
Flot d'exécution

1. Une erreur se produit dans une méthode.
2. La méthode crée un objet exception et l'émet.
3. Le *runtime* cherche quelqu'un capable de la traiter.



Traitement de l'exception

1. Le *runtime* cherche dans la pile d'appels, une méthode capable de traiter l'exception.
2. Si l'exception n'est pas traitée, l'exécution se termine.



Motivation

1. Séparation du traitement d'erreurs et du code.
2. Propagation d'erreurs.
3. Groupement et différenciation des types d'erreurs.

Séparation du traitement d'erreurs et du code

Exemple: Lecture du contenu d'un fichier.

```
lire_fichier  {
  // ouvrir le fichier;
  // déterminer sa taille;
  // allouer la mémoire
  // charger le fichier dans la mémoire;
  // fermer le fichier
}
```

Traitement classique d'erreurs (1/2)

```
errorCodeType readFile {
      initialize   errorCode = 0;
     // ouvrir le fichier;
     if (theFileIsOpen) {
         // déterminer sa taille;
         if (gotTheFileLength) {
             // allouer la mémoire
             if (gotEnoughMemory) {
                 // charger le fichier dans la mémoire;
                 if (readFailed) {errorCode = -1;}
             } else {errorCode = -2;}
         } else {errorCode = -3;}
 (...)
```

Traitement classique d'erreurs (2/2)

```
(...)
        // fermer le fichier
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
```

Traitement d'erreurs avec des exceptions

```
readFile {
    try {
        // ouvrir le fichier;
        // déterminer sa taille;
        // allouer la mémoire
        // charger le fichier dans la mémoire;
        // fermer le fichier
    } catch ( fileOpenFailed ) {
        doSomething;
    } catch ( sizeDeterminationFailed ) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch ( readFailed ) {
        doSomething;
    } catch ( fileCloseFailed ) {
        doSomething;
    }
}
```

Propagation d'erreurs

```
method1 {
      call  method2;
}

method2 {
      call  method3;
}

method3 {
      call   readFile ;
}
```

- Supposez que les erreurs de lecture de fichier n'intéressent que **method1()**.

Propagation d'erreurs

Approche classique

```
method1 {
  errorCodeType error;
  error = call method2;
  if (error)
    doErrorProcessing;
  else
    proceed;
}
```

```
errorCodeType method2 {
  errorCodeType error;
  error = call method3;
  if (error)
    return error;
  else
    proceed;
}
```

```
errorCodeType method3 {
  errorCodeType error;
  error = call readFile;
  if (error)
    return error;
  else
    proceed;
}
```

Propagation d'erreurs

Avec les exceptions

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```

Groupement et différentiation de types d'erreurs (1/2)

- Les exceptions sont des objets: il est possible de les grouper grâce à leur hiérarchie d'héritage.
- La classe **FileNotFoundException** n'a pas de descendant. Le code suivant ne traite que cette exception:

```
catch (FileNotFoundException e) {
    ...
}
```

Groupement et différentiation de types d'erreurs (1/2)

- Les exceptions sont des objets: il est possible de les grouper grâce à leur hiérarchie d'héritage.
- La classe **FileNotFoundException** n'a pas de descendant. Le code suivant ne traite que cette exception:

```
catch (FileNotFoundException e) {
    ...
}
```

- Il est possible d'écrire un code plus générique, capable de traiter toutes les exceptions de E/S:

```
catch (IOException e) {
    e.printStackTrace();    //Output goes to System.err.
    e.printStackTrace(System.out);   //Send trace to stdout.
}
```

Groupement et différentiation de types d'erreurs (2/2)

- Il est aussi possible d'écrire un code plus (trop) générique:

```
catch (Exception e) {     //A (too) general exception handler
    ...
}
```

Exceptions : émission

```
void Fichier :: ouvrir (const char* nom)
{
    if (nom_invalide(nom)) {
        throw erreur;
    }
}
```

- Il est possible d'émettre n'importe quel type d'objet! (par exemple un **enum** prenant un nombre fini de valeurs d'erreur);
- On émet généralement des instances de classes particulières.

Exceptions — émission

- On utilise en général des classes structurées par héritage (parfois multiple) comme exceptions. Voici des classes standards:

```
class exception; // namespace std, fichier exception
    class bad_cast       : public exception {...};
    class bad_alloc      : public exception {...};
    class runtime_error  : public exception {...};
        class overflow_error ...
    class logic_error    : public exception {...};
```

- Il est possible de créer un objet temporaire d'un type quelconque (disons **Exception**) par appel au constructeur par défaut (**throw Exception()**).

Exceptions — capture

Un bloc **try ... catch** signale qu'on va capturer certaines exceptions:

```
try {
    ...
}
catch ( range_error  re ) {
     cout << re.what();
     exit (1);
}
```

Si une exception de type **range_error** survient dans le bloc, on entrera dans le gestionnaire d'exception **catch** correspondant.

Exceptions — capture

- Il est possible de définir plusieurs gestionnaires à la suite:
- Les gestionnaires d'exception sont considérés dans l'ordre de leur déclaration;
- L'héritage compte: une exception est capturée dès qu'un gestionnaire capturant un type plus général est considéré;
- Il est possible de capturer une exception par valeur, par référence, ...

Exceptions — capture

En gros, le test des gestionnaires d'exception fonctionne comme un passage de paramètres.

- Il est possible de capturer toutes les exceptions par un gestionnaire **catch(...)**.

Exceptions — capture

```
try {
    ...
}
catch ( range_error )   { // capture anonyme
    throw;                // re-emission
}
catch ( bad_alloc  ba) { // capture par valeur
    cout << ba.what(); // bad-alloc::what
 }
catch (exception& e) { // capture par ref
    cout << e.what(); // appel polymorphe
}
catch  (...)  {
    cout << "inattendu" << endl;
    exit (1);
}
```

Exceptions — spécifications

Il est possible de préciser dans l'en-tête de fonction la liste des exceptions (susceptibles d'être) émises:

```
class Fichier {
public :
  class IOException {...};
  void open (const string &) throw(IOException);
};
```

La fonction peut émettre *seulement* des exceptions des types considérés.

Exceptions — destruction

```
struct Toto {
    ~Toto () {std::cout << "!" << std::flush;}
};

void rec (int i)
{
    Toto toto;

    if (i == 0)
        throw Exception();
    else
        rec (i - 1);
}

rec(10); // affichage?
```

Exceptions — conseil

Dans l'idéal il faut garantir que:

- Les constructeurs (par recopie, etc.) se comportent raisonnablement en cas d'exception (or l'instruction `new` peut générer une exception `bad_alloc`);
- Les ressources sont correctement désallouées en cas de sortie prématurée d'un bloc:

```
// l'exemple classique
{
    File f;
    f.open ("toto.txt");
    // ...
    // et si une exception survient ici??
    f.close ();
}
```

Emission

Exemple

```
public void loadFile(String fname) throws Exception {

  if ((fname==null||(fname.equals(""))) {
    Exception e = new Exception("pas de fichier"); // Creation de l'objet
    throw(e);  // Emission de l'Exception

    // syntaxe alternative:
    throw new Exception(" ");

    // Interruption de la methode
  }
  ...
}
```

Traitement

L'appelant de la méthode doit:

1. soit la renvoyer:

```
public void test() throws Exception {
   loadFile("truc.txt");
}
```

2. soit la traiter:

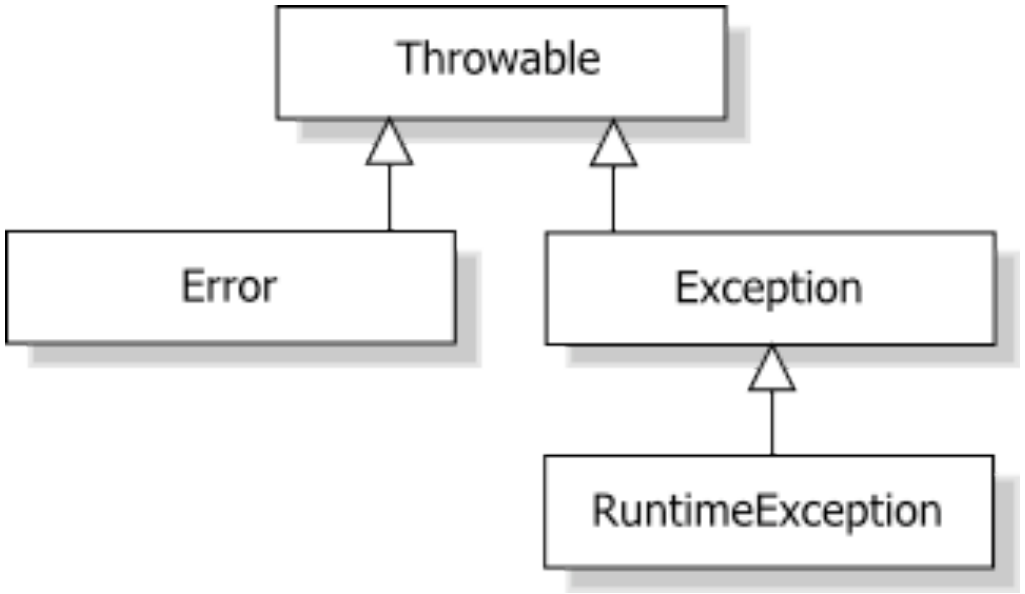
```
public void test() {
  try{
     loadFile("truc.txt");
     System.out.println("OK");
  } catch (Exception e) {
     System.err.println("Erreur de chargement: "+fe);
}
```

Catégories d'exception

En Java, il existe 3 catégories d'exceptions:

- Les exceptions vérifiées.
- Les exceptions de *runtime*.
- Les erreurs.

Exceptions de *Runtime* et Erreurs



1. Les Exceptions sous classes de **RuntimeException** ne nécessitent pas de déclaration **throws** ou d'un block **try - catch** car elles peuvent être déclenchées par la plupart des actions de la machine virtuelle.
2. Les Erreurs ne peuvent pas être traitées.

Exemples d'exceptions

- Exception
  - FileNotFoundException
  - ClassNotFoundException
- RuntimeException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ClassCastException
- Error
  - ClassFormatError
  - VerifyError

Exceptions personnalisées

Pour créer d'autres classes d'exception, il suffit de créer une sous-classe de la classe **Exception** et de redéfinir un de ses constructeurs.

```
public class MyException extends Exception {
  public MyException() {
    super("No details");
  }
}
```

Il est aussi possible d'y ajouter des attributs et des méthodes d'accès si nécessaire.

Exceptions multiples

S'il y a plusieurs Exceptions récupérables il faut plusieurs blocs de traitement (**catch**), du plus spécifique au plus général:

```
public void foo(){
  try {
     loadFile ("truc.txt");
     System.out. println ("Bon format");

  } catch (MyException fe) {
     System.out. println ("Mauvais format :"+fe);

  } catch (Exception e) {
     System.out. println ("Exception :"+e);
  }
}
```

Finally (1/2)

```
List<String> lines;
// ...
try {
    out = new PrintWriter(new FileWriter("Output.log"));
    for (String l : lines) {
        out.println(l);
    } catch (FileNotFoundException e) {
    // ...
    } catch (IOException e) {
    //...
    } finally {
        if (out != null)
            out.close();
     }
```

→ Le bloc **finally** est **toujours** exécuté.

Finally (2/2)

→ Pourquoi n'y a-t-il pas d'instruction **finally** en C++ ?

```
class File_handle {
  FILE* p;
public:
  File_handle(const char* n, const char* a)
    { p = fopen(n,a); if (p==0) throw Open_error(errno); }
  File_handle(FILE* pp)
    { p = pp; if (p==0) throw Open_error(errno); }
  ~File_handle() { fclose(p); }
  operator FILE*() { return p; }
// ...
};

void f(const char* fn)
{
  File_handle f(fn,"rw"); // open fn for reading and writing
  // use file through f
}
```

Definition

Une assertion est une expression booléenne qui indique une vérité sur laquelle un programme se base.
Si l'expression s'avère fausse, elle indique la présence d'un *bug* dans le programme.

Exemple

```
public class Stack {
  public int pop() {
    // precondition
    assert !this.isEmpty() : "Stack is empty";
    return stack[--num];
  }
}
```

Assertions et exceptions

Pouvoir et devoir

- Une Exception est un cas d'erreur qui peut être pris en charge par un programme. Une exception précise une situation qui peut se produire.
- Une Assertion violée est un cas d'erreur où il faut tout arrêter car il ne sera pas possible de continuer.
- Une assertion précise une condition qui doit être respectée pour éviter des situations qui ne doivent pas se produire.

catégories d'assertions

On distingue les catégories d'assertions suivantes :

- un **invariant de classe** spécifie l'état de tout objet de la classe.
- une **précondition** de méthode spécifie les conditions sur les paramètres (et eventuellement l'état de l'objet) qui permettent l'utilisation correcte de la méthode.
  par exemple, la methode substring(int) n'accepte que des entiers positifs
- une **postcondition** de méthode spécifie le résultat d'une méthode (en terme de retour ou de changements sur l'objet receveur)
- un **invariant de boucle** spécifie une propriété vraie tout au long de l'exécution de la boucle

Programmation par contrats : vision simplifiée

Bien que l'on puisse placer des assertions à de nombreux endroits d'un programme, elles sont souvent utilisées dans le cadre de la programmation par contrat.

Le contrat est un ensemble d'assertions publiques qui spécifie :

- les parties,
- les devoirs et droits de chacune
- et les conditions d'application.
- Le code client (appelant) doit satisfaire la précondition des methodes qu'il appelle
- Le code fournisseur (appelé) doit satisfaire la postcondition de ses méthodes et l'invariant de classe

Dans les langages qui ne supportent pas nativement la programmation par contrat, on utilise des commentaires pour le déclarer et des assertions non-spécifiques pour le vérifier à l'exécution.

Syntaxe

- Java:

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

- C++:

```
#include <assert.h>
// ...
assert(boolean-expression);
```

Syntaxe

En Eiffel, en plus de l'expression **check** *assertion* **end** on distingue les assertions suivantes :

- Préconditions : **require** *assertion*
- Postconditions : **ensure** *assertion*
- Invariants (de classe ou de boucle) : **invariant** *assertion*

Ces assertions (à l'exception de l'invariant de boucle) font automatiquement partie de la documentation du code car elles indiquent comment s'en servir et ce que l'on peut en attendre.

Contraintes de sous-typage lors de la redéfinition

Rappel du chapitre 5 : Un type doit respecter le contrat de son super type.
Si une méthode est redéfinie :

1. elle doit être accessible par le client.
2. les paramètres ne peuvent être "plus précis" que ceux attendus pour pouvoir accepter ceux du client,
   $\Rightarrow$ super-types : contravariance ou invariance

Contraintes de sous-typage lors de la redéfinition

Rappel du chapitre 5 : Un type doit respecter le contrat de son super type.
Si une méthode est redéfinie :

1. elle doit être accessible par le client.
2. les paramètres ne peuvent être "plus précis" que ceux attendus pour pouvoir accepter ceux du client,
   $\Rightarrow$ super-types : contravariance ou invariance
3. le type de retour ne peut être "moins précis" que celui attendu pour être exploité par un client,
   $\Rightarrow$ sous-types : covariance ou invariance

Contraintes de sous-typage lors de la redéfinition

Un type doit respecter le contrat de son super type.
Si une méthode est redéfinie :

- on ne peut exiger plus du client
  $\Rightarrow$ précondition plus faible ou identique
- on ne peut assurer (fournir) moins
  $\Rightarrow$ postcondition plus forte ou identique

Invérifiable dans le cas général. Le moyen le plus simple est d'hériter des assertions et de contraindre la redéfinition. C'est le cas en Eiffel :

- **require else** *assertion* construit une nouvelle précondition en ajoutant $\vee$*assertion* à la précondition héritée.
- **ensure then** *assertion* construit une nouvelle postcondition en ajoutant $\wedge$*assertion* à la postcondition héritée.

Exceptions / assertions

On peut considérer :

- qu'une *erreur* d'algorithme (accès contenu d'un conteneur vide, ...) doit être corrigée — pas question de laisser une boucle accéder à un élément invalide et de s'en tirer avec un **catch**;
- que des exceptions peuvent survenir principalement dans l'interaction avec l'utilisateur (action non valide, consommation mémoire, accès fichier...). Dans ce cas il n'y a pas d'autre choix que d'interrompre l'exécution classique et signaler l'exception.

A vous de vous faire votre avis sur la question!

Exceptions / assertions

A propos, en C++ vous pouvez définir votre propre version d'assert en utilisant les exceptions:

```
#ifdef NO_PRECONDITION_CHECKING
    define REQUIRE(condition)
#else
#   define REQUIRE(condition)\
        if (! (condition))\
            throw PreconditionViolation\
              (__FILE__, __LINE__, #condition);
#endif
```

Exemple (1/2)

La classe CarList

```
CarList :: CarList(const int max) {
    _max = max;
    _nb = 0;
    _cars = new RailroadCar*[max];
  }
void CarList :: add(RailroadCar* c) {
    assert (_nb < _max);

    _cars[_nb] = c;
    _nb++;
}
```

Exemple (1/2)

La classe CarList

```
CarList :: CarList(const int max) {
    _max = max;
    _nb = 0;
    _cars = new RailroadCar*[max];
  }
void CarList :: add(RailroadCar* c) {
    assert(_nb < _max);

    _cars[_nb] = c;
    _nb++;
}
```

→ Que se passe-t-il si **nb** >= **max** ?

Exemple (2/2)

Le programme principal

```
using namespace std;
int main() {
  Engine e;
  Train t(e);

  t.getCarList().add(new Coach(700, 20));
  t.getCarList().add(new Coach(300, 50));
  t.getCarList().add(new FreightCar(1000, 2000));

  for (int i = 1; i < 10 ; ++i) {
    t.getCarList().add(new Coach(700*i, 20));
  }

  std::cout << "Weight : " << t.totalWeight() << endl;;
}
```

Résumé

- Les exceptions sont des erreurs prévisibles, qui peuvent être traitées.
- Les assertions assurent le fonctionnement correct du programme. Une assertion fausse est une erreur d'implémentation.

Un peu de lecture...

- *Design by Contract*. – Bertrand Meyer. ISBN: 0130889210. Prentice Hall. Décembre 2000.