Conception objet en Java avec BlueJ
une approche interactive

11. Gestion des erreurs

David J. Barnes, Michael Kölling
version française: Patrice Moreaux

Concepts clés étudiés

- Programmation défensive
  - anticiper ce qui pourrait mal se passer
- Gestion et déclenchement des exceptions
- Notification d'erreurs
- Gestion de fichiers simple

Quelques situations causes d'erreurs

- Implantation incorrecte
  - non conforme à la spécification
- Requête inappropriée sur un objet
  - exemple: indice non valide
- État inconsistant ou inapproprié
  - exemple: lors d'une extension de classe

Pas toujours une erreur du programmeur !

- les erreurs proviennent souvent de l'environnement
  - URL entrée non valide
  - interruption réseau
- la gestion de fichiers est particulièrement génératrice d'erreurs
  - fichiers manquant
  - permissions non adaptées

Explorer des situations génératrices d'erreurs

- Expérimenter des situations d'erreurs avec les projets *address-book* (carnet d'adresses).
- Deux aspects:
  - signaler les erreurs
  - Gérer les erreurs

Programmation défensive

- interaction client-serveur
  - un serveur doit-il supposer que les clients sont «corrects»?
  - ou au contraire qu'ils sont *a priori* «hostiles» ?
- implique des implantations très différentes

Problèmes à étudier

- Niveau de vérification par le serveur lors des appels de méthodes ?
- Comment signaler les erreurs ?
- Comment un client peut-il anticiper une erreur ?
- Comment un client gère-t-il une erreur ?

Un exemple

- créez un objet AddressBook.
- Essayez de supprimer une entrée
- Résultat: une erreur d'exécution
  - à qui la faute ?
- Anticipation et prévention préférables à «chercher le coupable»

Valeurs des arguments

- Les arguments sont la principale faiblesse des objets serveur
  - Les arguments du constructeur initialisent l'état de l'objet
  - les arguments de méthode influencent souvent le comportement
- La vérification des arguments est une des mesures défensives

Vérification de la clé

```
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

Signaler une erreur de serveur

- Comment signaler des arguments incorrects ?
  - à l'utilisateur ?
    - il y a-t-il un utilisateur humain ?
    - peut-il résoudre le problème ?
  - à l'objet client ?
    - renvoyer une valeur de diagnostic
    - *déclencher (lancer, lever) une exception.*

Renvoyer un diagnostic

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                        (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Réponses du client

- Tester la valeur de retour
  - essayer de corriger l'erreur.
  - empêcher une erreur de programme
- Ignorer la valeur de retour
  - impossible de l'empêcher
  - risque de provoquer une erreur de programme
- Les exceptions sont préférables

Principes du déclenchement d'exception

- Fonctionnalité spécifique au langage
- Pas de valeur de retour spéciale
- Les erreurs ne peuvent être ignorées par le client
  - le flot d'exécution normal est interrompu
- Incite à des actions spécifiques de récupération sur erreur

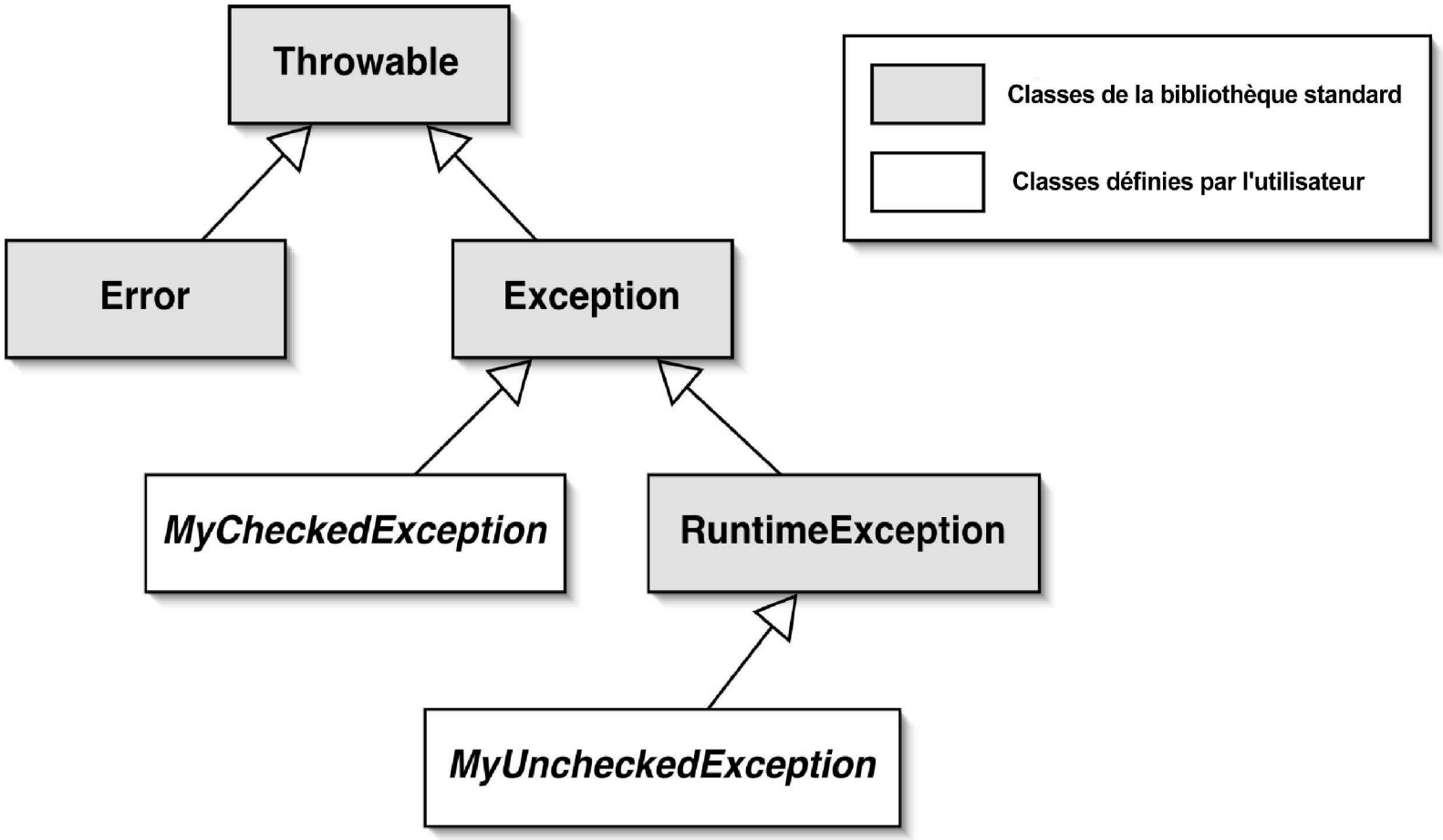
Déclencher une exception (1)

```
/**
 * Cherche un nom ou un numéro de téléphone et renvoie
 * les coordonnées correspondantes.
 * @param key Le nom ou le numéro à chercher.
 * @return Les coordonnées correspondant à la clé,
 *         ou 'null' si pas trouvées.
 * @throws NullPointerException si la clé est 'null'.
 */
public ContactDetails getDetails(String key)
{
    if(key == null){
        throw new NullPointerException(
                    "clé 'null' dans 'getDetails'");
    }
    return (ContactDetails) book.get(key);
}
```

Déclencher une exception (2)

- Un objet exception est créé :
  - `new ExceptionType("...");`
- L'objet exception est déclenché :
  - `throw ...`
- Documentation Javadoc :
  - `@throws ExceptionType cause`

Hiérarchie des classes d'exception



Catégories d'exceptions

- Exceptions sous contrôle
  - sous-classes de **`Exception`**
  - pour anticiper les erreurs
  - lorsqu'une récupération sur erreur est possible
- Exceptions hors contrôle
  - sous-classes de **`RuntimeException`**
  - pour les erreurs non prévisibles
  - lorsqu'une récupération est improbable

Effet d'une exception

- La méthode qui déclenche se termine prématurément.
- Pas de valeur retournée.
- L'exécution ne reprend pas au point d'appel.
  - donc le client ne peut ignorer l'exception
- Un client peut *capturer* ('catch') une exception.

Exceptions hors contrôle

- Usage non contrôlé par le compilateur.
- Cause la terminaison du programme si non capturée.
  - C'est le cas en général.
- Exemple typique: **`IllegalArgumentException`**

Vérification des arguments

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                            "clé 'null' dans
'getDetails'");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                           "clé vide passée à
'getDetails'");
    }
    return (ContactDetails) book.get(key);
}
```

Empêcher la création d'objet

```
public ContactDetails(String name, String phone, String address)
{
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                "Le nom ou le téléphone doit être non vide.");
    }
}
```

Gestion des exceptions

- Les exceptions sous contrôle doivent être capturées.
- Le compilateur vérifie leur bonne utilisation.
  - dans le client et dans le serveur.
- Bien utilisées, les erreurs peuvent être «récupérées».

La clause **`throws`**

- Le méthodes qui peuvent déclencher une exception sous contrôle doivent comporter une clause **`throws`**:

```
public void saveToFile(String destinationFile)
    throws IOException
```

Le bloc `try` (1)

- Les clients capturent l'exception et protègent l'appel dans un bloc **`try`** :

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
```

Le bloc `try` (2)

1. Exception déclenchée ici

2. Contrôle transféré ici

```
try{
     addressbook.saveToFile(filename);
     tryAgain = false;
}
catch(IOException e) {
     System.out.println("Impossible de sauvegarder
                         dans " + filename);
     tryAgain = true;
}
```

Capturer plusieurs exceptions

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Action sur exception end-of-file.
    ...
}
catch(FileNotFoundException e) {
    // Action sur exception file-not-found.
    ...
}
```

La clause `finally` (1)

```
try {
    Protéger une ou plusieurs instructions ici.
}
catch(Exception e) {
    Informer et/ou corriger sur exception ici.
}
finally {
    Actions toujours exécutées, avec ou sans
    exception.
}
```

La clause `finally` (2)

- Une clause **`finally`** est exécutée même si une instruction return est exécutée dans les clauses **`try`** ou **`catch`**.
- Une exception non capturée ou propagée subsiste même avec une clause **`finally`**.

Définir de nouvelles exceptions (1)

- Étendre **`Exception`** or **`Runtime-Exception`**.
- Définir de nouveaux types pour fournir une meilleure information.
  - inclure un compte rendu et/ou des informations de récupération.

```
public class NoMatchingDetailsException extends Exception
{
    private String key;

    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }

    public String getKey()
    {
        return key;
    }

    public String toString()
    {
        return "No details matching '" + key +
               "' were found.";
    }
}
```

Définir de nouvelles exceptions (2)

Récupération sur erreur

- Les clients devraient prendre en compte les notifications d'erreur.
  - test des valeurs retournées.
  - ne pas ignorer les exceptions.
- Inclure le code pour essayer de corriger l'erreur.
  - nécessite souvent une boucle.

Tentative de récupération

```
// Essayer de suavegarder le carnet d'adresses.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Impossible de sauvegarder dans " +
filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = autre_nom_de_fichier;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Signaler le problème et finir;
}
```

Éviter les erreurs

- Les clients peuvent souvent employer des méthodes de requête pour éviter les erreurs.
  - des clients plus fiables autorisent des serveurs plus « confiants ».
  - utiliser des exceptions hors contrôle.
  - Simplifie la logique du client.
- Peut accroître de couplage client-serveur.

Entrées/sorties texte

- Les entrées/sorties sont source d'erreurs.
  - elles mettent en jeu des interactions avec l'environnement extérieur.
- Le paquetage Java **`java.io`** gère les entrées/sorties.
- **`java.io.IOException`** est une exception contrôlée.

`Readers`, `Writers`, `Streams`

- **`Readers`** et **`Writers`** gèrent les entrées/sorties textuelles.
  - fondés sur le type **`char`** (caractère).
- **`Streams`** gère les données binaires.
  - fondés sur le type **`byte`** (octet).
- Le projet *address-book-io* fournit des exemples d'entrées/sorties texte.

Sortie de texte (1)

- Utilise la classe **`FileWriter`**.
  - ouvrir un fichier.
  - écrire dans le fichier.
  - fermer le fichier.
- Une erreur à un instant quelconque génère une exception **`IOException`**.

Sortie de texte (2)

```
try {
    FileWriter writer = new FileWriter("nom du fichier");
    while(il y a du texte à écrire) {
        ...
        writer.write(partie de texte suivante);
        ...
    }
    writer.close();
}
catch(IOException e) {
    il y a eu un problème pendant l'accès au fichier
}
```

Entrée de texte (1)

- Utilise la classe **`FileReader`**.
- Extension avec **`BufferedReader`** pour les entrées par ligne.
  - ouvrir le fichier.
  - lire à partir du fichier.
  - fermer le fichier.
- Une erreur à un instant quelconque génère une exception **`IOException`**.

Entrée de texte (2)

```
try {
    BufferedReader reader = new BufferedReader(
                            new FileReader("nom du fichier "));
    String line = reader.readLine();
    while(line != null) {
        utiliser la ligne
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    on ne trouve pas le fichier indiqué
}
catch(IOException e) {
    il y a eu un problème pendant la lecture ou à la fermeture
}
```

Résumé (1)

- Les causes d'erreurs d'exécution sont multiples.
  - appel inapproprié d'un client à un objet serveur.
  - serveur incapable de satisfaire à une requête.
  - erreurs de programmation dans le client et/ou le serveur.

Résumé (2)

- Les erreurs d'exécution génèrent souvent des abandons du programme.
- La programmation défensive anticipe les erreurs – dans le client et dans le serveur.
- Les exceptions fournissent un mécanisme pour signaler et gérer les erreurs.

Sommaire général