Programmation robuste et exceptions

LU3IN002 : Programmation par objets

L3, Sorbonne Université

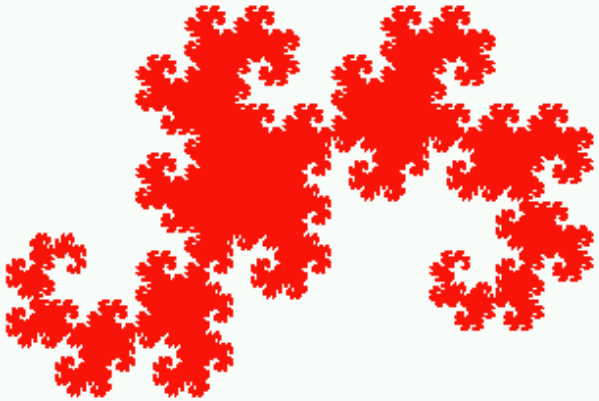
`https://moodle-sciences-23.sorbonne-universite.fr`

Antoine Miné

Cours 5

4 octobre 2023

Année 2023–2024



SORBONNE UNIVERSITÉ

LIP6

Plan du cours

- Cours 1, 2 & 3 : Introduction et bases de Java
- Cours 4 : Collections, itérateurs
- Cours 5 : Exceptions, tests unitaires
- Cours 6 : Design patterns I : Design Patterns structurels
- Cours 7 : Polymorphisme
- Cours 8 : Design patterns II : Design Patterns comportementaux
- Cours 9 : Interfaces graphiques (JavaFX)
- Cours 10 : Design patterns III : Design Patterns créationnels
- Cours 11 : Aspects fonctionnels de Java (lambdas)

Aujourd'hui :

- les exceptions en Java
- la programmation par contrats
- les tests unitaires, avec JUnit 4
- les annotations @ en Java

Motivation

Les erreurs dans les logiciels coûtent cher !

Quelques exemples :

- bug « Heartbleed » d'OpenSSL, 500 M$ (eWEEK, 2014) ;
- bug du contrôleur de vitesse de Toyota (2003–), 89 décès (CBSNews, 2010).

Variété de méthodes de vérification complémentaires :

- vérification statique (à la compilation) ou dynamique (test par exécution) ;
- méthodes empiriques ou formelles (basées sur la logique mathématique) ;
- assurance basée processus ou basée produit.

Dans l'industrie avionique critique, 70% du coût de développement concerne la vérification.

Tendance actuelle : vers des méthodes de plus en plus formelles.

La conception du langage peut aider.
Comparer l'effet d'accéder à un pointeur nul ou invalide :

- En C, C++ : comportement indéfini, tout peut arriver ;
- En Java : comportement bien défini, erreur rattrapable ;
- En OCaml : les erreurs de pointeur sont impossibles par construction.

Les exceptions

Exceptions

Mécanisme de traitement des comportements exceptionnels,
en particulier les erreurs.

Une exception peut être signalée :

- par le système, par une bibliothèque standard ;
- ou par l'utilisateur : instruction `throw`.

Quand une exception est signalée :

- le cours normal de l'exécution du programme (le flot de contrôle) est interrompu ;
- le programme saute à un gestionnaire, défini par l'utilisateur
  par la construction `try ... catch ... finally`.
  (ou le programme quitte avec un message d'erreur si aucun gestionnaire n'est présent)

En Java, une exception est un objet :

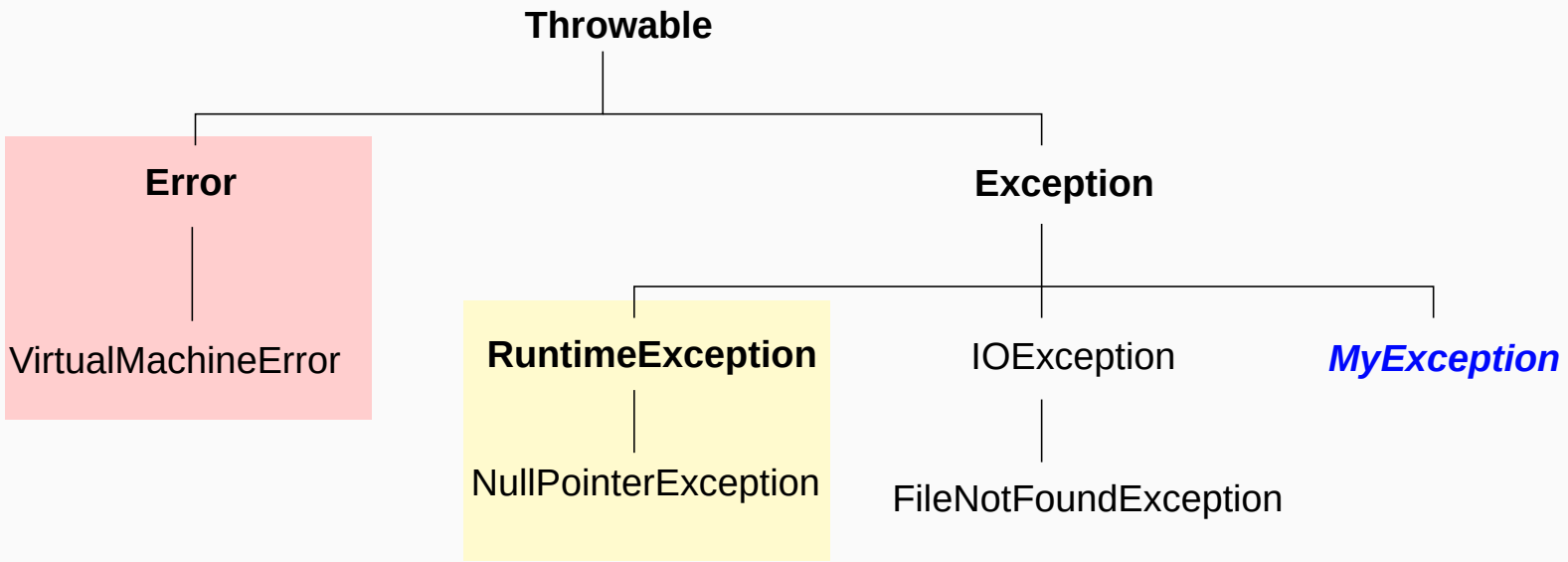
- dérivant de la classe `Throwable`, et généralement de sa sous-classe `Exception` ;
- pouvant porter des informations dans des attributs (nature de l'erreur, etc.) ;
- l'utilisateur peut définir des classes d'exception personnalisées.

Quelques exceptions utiles

Quelques exceptions prédéfinies que vous rencontrerez souvent :

- dans `java.lang`
  - `NullPointerException`
    tentative d'accès à `null` (attribut, appel de méthode)
  - `ArrayIndexOutOfBoundsException`
    accès à un tableau en dehors de ses bornes (< 0 ou $\geq$ length)
  - `ClassCastException`
    conversion dans un type incompatible avec le type dynamique
  - `ArrayStoreException`
    stocker dans un tableau un objet incompatible avec le type du tableau
  - `ClassNotFoundException`
    fichier `.class` non trouvé (rappel : les classes sont chargées durant l'exécution)
  - `ArithmeticException`
    division par zéro entière, etc. (mais pas les erreurs de flottants !)
- dans `java.io`
  - `IOException`
    erreur d'entrée-sortie (fichier)
  - `FileNotFoundException`
    fichier non trouvé, cas particulier d'`IOException`

Hiérarchie des exceptions



- `Error` : racine des erreurs graves du système ;
- `RuntimeException` : racine des exceptions non vérifiées à la compilation ;
- `MyException` : exception utilisateur, qui hérite de `Exception`.

  Dans la suite, les exceptions que nous définirons dériveront de `Exception` mais pas de `RuntimeException`.

Gestion des exceptions : mécanisme *try-catch*

```
try {
    // code protégé
}
catch (IOException var) {
    // code du gestionnaire
}
// code suivant, non protégé
```

Trois cas de figure :

1. si le code protégé s'exécute complètement sans exception :
   $\Longrightarrow$ le code du gestionnaire est ignoré,
   l'exécution reprend au début du code, non protégé, suivant le bloc `catch` ;
2. si une exception de classe `IOException` ou dérivée interrompt ce code :
   $\Longrightarrow$ l'exécution reprend au début du code du gestionnaire `catch`,
   après le gestionnaire, le code suivant, non protégé, s'exécute ;
3. si une exception ne dérivant pas de `IOException` interrompt le code protégé :
   $\Longrightarrow$ le code du gestionnaire et le code suivant sont sautés,
   le système continue la recherche d'un gestionnaire adapté...

Exemple de *try-catch*

exemple : lecture d'un fichier

```
BufferedReader b = new BufferedReader(new FileReader("file.txt"));
try {
    while (true) {
        String s = b.readLine();
        if (s == null) break;
        System.out.println("J'ai lu : " + s);
    }
    System.out.println("Fin normale");
}
catch (IOException e) {
    System.out.println("Une erreur");
}
b.close();
```

Lecture d'un fichier ligne par ligne, en signalant les erreurs de lecture.

Note : `BufferedReader` sert à lire ligne par ligne ; `FileReader` ne sait lire qu'un caractère à la fois.

En cas d'exception ou de fin normale (`readLine` renvoie `null`)
le fichier est fermé avec `b.close()`.

Note : le fait que `readLine` peut signaler une `IOException` est indiqué dans la documentation de l'API Java, par :

```
public String readLine() throws IOException
```

`new FileReader(...)` peut aussi signaler une `IOException`, mais celle-ci n'est pas gérée par notre exemple...

Exceptions comme objets

Une exception est un objet, avec des constructeurs et des méthodes.

Quelques méthodes utiles, héritées de `Throwable` :

- `String getMessage()`
  retourne un message d'erreur détaillé
- `String toString()`
  retourne un message d'erreur succinct
- `void printStackTrace()`
  affiche la pile complète d'exécution menant à l'exception
- `Throwable getCause()`
  permet le chaînage d'exceptions (exception durant le traitement d'une exception)

exemple : affichage d'une erreur

```
try {  ... }
catch (IOException e) { System.out.println("I/O Error: " + e.getMessage()); }
```

Gestionnaires d'exceptions multiples, héritage

exemple : exceptions liées aux fichiers

```
try {
    f = new FileReader();
    int r = f.read();
    f.close();
}
catch (FileNotFoundException e)  { ... }
catch (IOException e)            { ... }
catch (Exception e)              { ... }
```

Un bloc `try` peut avoir plusieurs clauses `catch` :

- `catch (C e)` accepte l'exception `obj` si `obj instanceof C` est vrai ;
- en cas d'exception, les clauses sont examinées dans l'ordre ;
- seule la première clause ayant un type compatible est exécutée ;
- si aucune clause avec le bon type n'est trouvée,
  la recherche continue à partir du bloc `try catch` immédiatement englobant ;

$\Longrightarrow$ permet de regrouper la gestion des exceptions,
en traitant du plus spécifique au plus général.

L'exemple ci-dessus gère toutes les exceptions de type `Exception`,
avec des traitements particuliers pour `IOException` et `FileNotFoundException`.

En fait, seules les exceptions de type `Error` ne sont pas gérées (elles n'héritent pas d'`Exception`) ;
c'est normal : `Error` correspond aux erreurs graves que le programme ne peut pas gérer en règle générale.

Gestion d'exceptions multiples en Java 7

en Java < 7

```
try { ... }
catch (NullPointerException e)       { code }
catch (ClassCastException e)         { code }
catch (IndexOutOfBoundsException e) { code }
```

en Java $\geq$ 7

```
try { ... }
catch (NullPointerException | ClassCastException | IndexOutOfBoundsException e) { code }
```

Java 7 introduit une nouvelle notation pour regrouper le traitement d'exceptions de plusieurs classes indépendantes avec le même gestionnaire : le symbole `|`.

Évite :

- la duplication du code de gestionnaire ;
- le recours à la capture d'exceptions trop générales.

  e.g., capturer un parent de `NullPointerException`, `ClassCastException`, etc.
  i.e., `catch (Exception e) { ...}`
  $\Longrightarrow$ cela capture aussi d'autres exceptions non voulues, comme `ArithmeticException`

Le type statique de e sera la classe parent englobant tous les cas possibles, ici `RuntimeException`.

Délégation implicite à l'appelant

exemple : addition

```
private int asInt(Object o)
{
    Integer i = (Integer) o;  // peut signaler une exception...
    return o.intValue()
}

public Object add(Object o1, Object o2)
{
    try {
        return new Integer(asInt(o1) + asInt(o2));
    }
    catch (ClassCastException c) {
        // ... qui est traitée ici
        return new Integer(0);
    }
}
```

Si une méthode reçoit une exception mais ne la traite pas,
l'exception est retransmise à l'appelant, et ainsi de suite,
jusqu'à trouver un gestionnaire d'exception.

$\Longrightarrow$ les erreurs de conversion de type signalées dans `asInt` sont en réalité traitées dans `add`.

Absence de traitement : remontée au *top-level*

Si, en remontant la pile d'appel des méthodes, la JVM ne trouve pas de gestionnaire d'exception adéquat, le programme est interrompu et l'exception est affichée à l'écran.

exemple

```
package nul;

public class NullMain {
    private String s;

    public int f()
    { return s.length() + 1; }

    public static void main(String[] args)
    { System.out.println((new NullMain()).f()); }
}
```

sortie

```
Exception in thread "main" java.lang.NullPointerException
        at nul.NullMain.f(NullMain.java:6)
        at nul.NullMain.main(NullMain.java:8)
```

La sortie indique la pile d'appels complète, ce qui est très utile au débogage.

(affichage similaire à celui de `printStackTrace()`)

Signaler des exceptions : le mot-clé throw

Un programme peut signaler des exceptions : avec `throw exception`.

classe vecteur 2d immuable

```
public class Vector2D {
    private final double x, y;

    public Vector2D(double x, double y) { this.x = x; this.y = y; }

    public Vector2D normalize() {
        double d = Math.sqrt(x*x + y*y);
        if (d <= 0.0001) {
            throw new IllegalArgumentException("vector too small");
        }
        return new Vector2D(x / d, y / d);
    }
}
```

L'exception se construit comme tout objet, avec `new` et un appel au constructeur.

`IllegalArgumentException` est une classe prédéfinie de `java.lang` pour signaler un argument invalide.
Ici, l'exception est créée dans la méthode `normalize`, mais n'est pas capturée dans la méthode.

En pratique, il est souvent utile de définir et d'utiliser
ses classes d'exception personnalisées (voir plus loin).

Délégation explicite avec `throw`

exemple

```
try {
    f = new FileReader("file.txt");
    ...
    f.close();
}
catch (Exception e) {
    if (f != null) f.close();
    System.out.println(e.getMessage());
    throw e;
}
```

Après le traitement, l'exception peut être signalée à nouveau dans le gestionnaire.
Elle sera donc également traitée par le gestionnaire englobant.

Le chaînage d'exceptions permet d'associer une exception à une autre ;
nous remplaçons pour cela `throw e` par :

```
throw new Exception("Error in my class", e);
```

- nous lançons une nouvelle exception, qui masque l'exception originale e
- l'exception originale e peut être retrouvée avec `getCause()`.

Important : ne pas utiliser un gestionnaire pour cacher l'exception !

pas de : `catch (Exception e) { /* rien */ }`

si le problème ne peut pas être résolu dans la méthode, déléguer avec `throw`.

Déclaration des exceptions et échappement

Chaque méthode doit lister avec `throws` les exceptions qui peuvent échapper :

- les exceptions signalées directement avec `throw`,
  ou indirectement par une méthode appelée,
- et qui ne sont pas gérées par un `catch` dans la méthode elle-même.

On utilise la syntaxe `throws exception1, exception2, ...` dans la signature.

(ne pas confondre le mot-clé `throw` et le mot-clé `throws` !)

exemple

```
public void f() throws IOException;

public void g() throws IOException {
    f(); // IOException possible et non rattrapée
}

public void h() {
    try { f(); } // IOException possible et rattrapée
    catch(IOException e) { ... }
}
```

Avantages :

- documente la méthode avec les cas d'échec possibles ;
- le compilateur vérifie la compatibilité des `throw` et `throws`,
  et s'assure que seules les exceptions autorisées peuvent s'échapper.

Les exceptions non vérifiées : `RuntimeException`

La déclaration des exceptions d'une méthode avec `throws`
ne concerne pas les classes dérivées de `RuntimeException`.

Pour elles, l'existence d'un gestionnaire n'est donc pas vérifié à la compilation !

Exemples : `ClassCastException`, `IllegalArgumentException`,
`NullPointerException`, `ArrayIndexOutOfBoundsException`

Justification :

Ces exceptions peuvent être signalées par de nombreuses constructions du langage,
(« dans le cours normal de la JVM »)
imposer de les traiter exhaustivement a été jugé trop contraignant.

Ce choix est controversé : il rend la vérification statique des exceptions moins utile.

Exemple : pas d'assurance que les références `null` seront correctement gérées.

Mécanisme *try-catch-finally*, mot-clé `finally`

exemple

```
BufferedReader b = null;
try {
    b = new BufferedReader(new FileReader(path));
    return b.readLine();
}
catch(IOException e)  { return ""; }
finally               { if (b != null) b.close(); }
```

Si elle est présente, la clause `finally` est **toujours exécutée** :

- s'il n'y a pas d'exception : juste avant de sortir du bloc `try`,
  même si le bloc `try` sort de la méthode par `return` !
- si une exception est traitée par une clause `catch`,
  juste avant de sortir de la clause `catch`,
  même si la clause `catch` s'achève par un `return` ou un `throw` !
- si une exception n'est pas traitée par une clause `catch`,
  juste avant de propager l'exception au bloc englobant ou à l'appelant.

$\Longrightarrow$ utile pour ajouter du code de « nettoyage ».
(fermer un fichier, libérer de la mémoire, etc.)

Mécanisme *try-finally, try-with-resource*

en Java < 8

```
BufferedReader b = new BufferedReader(new FileReader(path));
try     { return b.readLine(); }
finally { if (b != null) b.close(); }
```

Un bloc `try-finally` sans `catch` est aussi très utile :

- il délègue complètement le traitement de l'exception à l'appelant,
- et s'assure que les ressources allouées dans la méthode sont bien libérées.

en Java $\geq$ 8

```
try (BufferedReader b = new BufferedReader(new FileReader(path)))
{ return b.readLine(); }
```

Java 8 introduit une syntaxe spéciale pour ce motif :

```
try (Classe var = expr) { ... }
```

la classe doit obéir à l'interface `java.lang.AutoCloseable`
qui définit une méthode `public void close()`.

`var.close()` est automatiquement appelée en fin de bloc, qu'il y ait eu une exception ou non.

Classes d'exception personnalisées

Une exception est un objet comme un autre ;
elle peut être une instance d'une classe définie par l'utilisateur.

exemple

```
public class FileError extends Exception {
  private String filename;
  public FileError(String name)      { super("Cannot handle " + name); filename = name; }
  public String getFilename()        { return filename; }
  @Override public String toString() { return "error on " + filename; }
}
```

Pour être utilisable avec `throw`, il suffit d'hériter de `Throwable`.

ou d'une classe héritant de `Throwable`, généralement `Exception`

Avantages :

- classifier les types d'erreurs pour faciliter leur traitement ;

  `catch (FileError e)` où `e instanceof FileError`
  $\Longrightarrow$ créer une hiérarchie de classes d'erreurs est très utile

- distinguer les exceptions client des exceptions du système et des bibliothèques ;
- ajouter des attributs et des méthodes adaptées aux erreurs.

Programmation par contrats

Programmation par contrats : principes

But : programmation **modulaire robuste**.

Principe : définir un contrat entre

- une classe qui fournit une fonctionnalité ;
- et un client qui exploite cette fonctionnalité (i.e., appelle des méthodes).

Il précise :

- les règles que le client doit respecter pour utiliser la classe (contraintes) ;
- les garanties offertes par le fournisseur (sûreté) ;
- en ignorant les détails d'implantation (abstraction).

Utilisation : le contrat peut servir à :

- documenter, pour aider la conception des clients ;
- vérifier la correction du programme au fur et à mesure de son exécution ;
- tester la correction du programme avant son déploiement ;
- vérifier formellement la correction du programme avant son déploiement.

Pré-conditions, post-conditions, invariants

schéma du fournisseur

```
private t1 attr1, ..., attrN;  // invariants

public t m(t1 arg1, ..., tN argN) {
    // pré-conditions de m
    ... code ...
    // post-conditions de m
}
```

Un contrat spécifie pour chaque méthode :

- des **pré-conditions**

  i.e., des pré-requis pour le bon fonctionnement de la méthode,
  comme des conditions sur les arguments et sur l'état de l'objet (attributs) ;

- des **post-conditions**

  i.e., des garanties offertes par le fournisseur,
  comme des propriétés sur la valeur de retour et l'état de l'objet,
  et des relations entre l'état de l'objet avant et après la méthode ;

et pour la classe entière :

- des **invariants**

  i.e., des propriétés toujours vraies de l'état des objets de la classe.

Exemple de contrat : code et invariant

exemple : allocateur

```
public class ResourceAllocator {
    private int capacity;
    private int allocated = 0;
    // invariant : 0 <= allocated && allocated <= capacity

    public ResourceAllocator(int c) { capacity = c; }

    public int allocate(int n) {
        if (allocated + n > capacity) n = capacity - allocated;
        allocated += n;
        return n;
    }

    public void setCapacity(int newCapacity) { capacity = newCapacity; }
}
```

Une classe pour allouer des ressources : (ici, des unités entières)

- `capacity` : quantité totale de ressources disponibles ;
- `allocated` : quantité de ressources déjà allouées ;
- `allocate(n)` : alloue `n` ressources (ou moins, en cas de pénurie de ressources) ;
- `setCapacity(c)` : change la quantité totale de ressources disponibles.

Invariant : à tout instant, $0 \leq$ `allocated` $\leq$ `capacity`.

Exemple de contrat : pré et post-conditions

exemple : allocateur annoté

```
// invariant : 0 <= allocated && allocated <= capacity

public ResourceAllocator(int c) {
    // pré-condition : c >= 0
    capacity = c;
    // post-condition : capacity == c && allocated == 0
}

public int allocate(int n) {
    // pré-condition : n >= 0
    if (allocated + n > capacity) n = capacity - allocated;
    allocated += n;
    return n;
    // post-condition : 0 <= return <= old(n) && allocated == old(allocated) + return
}

public void setCapacity(int newCapacity) {
    // pré-condition : newCapacity >= allocated
    capacity = newCapacity;
    // post-condition : capacity == newCapacity
}
```

Les pré-conditions d'une méthode sont nécessaires pour assurer :

- les post-conditions de la méthode ;
- le maintient de l'invariant ;
- l'absence d'erreur à l'exécution grave, d'exception `RuntimeException` non vérifiée.

Contrats et modularité

fournisseur

```
public int allocate(int n) {
    // suppose les pré-conditions
    // suppose les invariants
    ... code ...
    // s'assure des post-conditions
    // s'assure des invariants
}
```

client

```
// s'assure de la pré-condition
alloc.allocate(10);
// suppose la post-condition
```

Séparation des responsabilités :

- le fournisseur suppose que la pré-condition est vraie en début de méthode,
- il doit faire en sorte que la post-condition est vraie en fin de méthode,
- et que l'invariant est maintenu ;
- le client fait en sorte que la pré-condition est vraie avant l'appel,
- il peut supposer que la post-condition et l'invariant sont vrais après l'appel,
- le client ne peut pas accéder à l'état directement pour invalider l'invariant ;
  (encapsulation : les attributs du fournisseur sont privés)

$\Longrightarrow$ chaque classe peut être vérifiée indépendamment des autres classes.

Typage comme contrat

La signature d'une méthode est en réalité déjà une forme de contrat.

`public A m(B arg)` signifie :

- pré-condition : `arg` référence un objet de classe `B` ou dérivée ;
- post-condition : `m` retourne une référence sur un objet de classe `A` ou dérivée ;
- pas d'exception signalée par `m` (sauf `RuntimeException`).

Plus généralement, une interface (ensemble de signatures) est un contrat de classe.

Avantage :

- vérification statique et automatique par le compilateur
  $\Longrightarrow$ assurance que le contrat est respecté durant toute exécution.

Limitation : ce type de contrat est peu expressif

- pas d'information sur les valeurs des variables ou des attributs ;
- pas d'indication si une référence peut être `null` ou pas ;

$\Longrightarrow$ généralement insuffisant pour exprimer les conditions d'utilisation du fournisseur
(e.g., pas d'argument `null`)

Vérification des contrats du fournisseur

Problème : s'assurer de la robustesse du fournisseur

- ne pas faire confiance au client ;
  $\Longrightarrow$ validation systématique des pré-conditions des méthodes,
  donc y compris en production,
  utilisation d'exceptions pour signaler la violation du contrat ;
- se convaincre que les méthodes du fournisseur sont correctes,
  i.e., maintiennent les invariants et assurent les post-conditions ;
  $\Longrightarrow$ test unitaire (voir un peu plus loin)
  seulement pendant la phase de mise au point du fournisseur.

Le client n'a rien à vérifier,
mais il doit être prêt à gérer les exceptions signalées par le fournisseur.

Note : le client a aussi parfois le rôle de fournisseur vis à vis d'un autre client, et a donc son propre contrat.

Création d'une hiérarchie d'exceptions

exceptions de l'allocateur

```
public class AllocatorException extends Exception {
    public AllocatorException(String msg) { super(msg); }
}

public class InvalidCapacity extends AllocatorException {
    public InvalidCapacity() { super("invalid capacity"); }
}

public class InvalidAlloc extends AllocatorException {
    public InvalidAlloc() { super("invalid allocation"); }
}
```

Bonne pratique :

- créer une classe d'exception dédiée aux fournisseurs d'un contrat ;
- puis créer des sous-classes par type précis d'erreur.

Avantages :

- isole les erreurs liées à ce contrat des autres types d'erreurs ;
- permet au choix un traitement général de toutes les erreurs du contrat, ou un traitement plus fin de sous-classes précises d'erreurs.

Vérification des pré-conditions du contrat

exemple : allocateur annoté

```
public ResourceAllocator(int c) throws InvalidCapacity
{
    if (c < 0) throw new InvalidCapacity();
    capacity = c;
}

public int allocate(int n) throws InvalidAlloc
{
    if (n < 0) throw new InvalidAlloc();
    if (allocated + n > capacity) n = capacity - allocated;
    allocated += n;
    return n;
}

public void setCapacity(int newCapacity) throws InvalidCapacity
{
    if (newCapacity < allocated) throw new InvalidCapacity();
    capacity = newCapacity;
}
```

- seules les pré-conditions sont matérialisées dans le code exécuté ;
- les post-conditions et les invariants doivent apparaître en documentation.
  (commentaires `javadoc` dans le code, `@return`)

Tests unitaires avec JUnit 4

Principes du test

Les programmes sont testés avant déploiement pour s'assurer de leur correction.

Plusieurs niveaux de test :

- **test unitaire**
  - vérifier indépendemment chaque méthode de chaque classe ;
  - les contrats guident l'élaboration des banques de tests ;
- **test d'intégration**
  - combiner et tester des morceaux d'application de plus en plus gros ;
  - vérifier la compatibilité des interfaces et des contrats.

On distingue deux catégories de propriétés testées :

- tests fonctionnels : le programme obéit à sa spécification, à son contrat ;
- tests non-fonctionnels : le programme n'a pas d'erreur fatale à l'exécution
  (spécification implicite, fournie par le langage et indépendante de ce que le programme doit calculer)

Bonne pratique : développement **piloté par les tests**

- ne pas tester au dernier moment, mais au fur et à mesure du développement ;
- écrire en premier les contrats, puis les tests, et écrire le code en dernier ;
- tester immédiatement après implantation (bonne motivation : le test est déjà disponible) ;
- effectuer des tests de non-régression après chaque ajout de fonctionnalité.

Présentation de JUnit 4

JUnit 4 : système de test pour Java

- chaque test est une méthode,
  - soit intégrée dans la classe à tester,
  - soit isolée dans une classe de test séparée (conseillé) ;
- langage puissant d'assertions pour exprimer les propriétés à tester :
  (comparaison entre le comportement attendu et celui effectivement observé)
- intégration avec Eclipse :
  - lancement par clic droit sur la classe → « Run As » → « JUnit Test » ;
  - ne pas oublier d'ajouter la bibliothèque `junit.jar`
    clic droit sur le projet → « Properties » → « Java Build Path » → « Libraries » → « Add Library... » → « JUnit » → « JUnit 4 »
  - visualisation graphique des résultats (barre verte si le test passe, rouge sinon) ;
- gestion de tests multiples ;
- possibilité de factoriser du code commun à plusieurs tests ;
- système très extensible.

Nous utilisons la version JUnit 4, basée sur les annotations Java `@Test`.

Voir : `http://junit.org/` (la dernière version est JUnit 5...)

Exemple de test

test de l'allocateur

```
package pobj.cours5.test;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class ResourceAllocatorTest {

    @Test public void testAllocate() {
        ResourceAllocator a = new ResourceAllocator(10);
        int r = a.allocate(6);
        assertEquals(6, r);
        assertEquals(6, a.getAllocated());
        assertEquals(4, a.getFree());
    }
}
```

Les méthodes de test :

- sont regroupées dans une classe séparée : `ResourceAllocatorTest`, d'un package séparé `pobj.cours5.test` ; (organisation plus rationnelle)
- sont publiques, non statiques, et marquées de l'annotation `@Test` ;
- contiennent un code exécutant chacune un cas de test reproduisible ;
- se terminent en testant les post-conditions et les invariants attendus.

Langage d'assertion : test de conditions

Assertions : méthodes statiques de `org.junit.Assert`.

il faut donc utiliser : `import static org.junit.Assert.`*méthode*

- `assertTrue (booléen)`
  `assertFalse (booléen)`
  vérifie qu'une expression booléenne est vraie, ou fausse ;
- `assertEquals (valeurAttendue, valeurCalculée)`
  `assertNotEquals (valeurAttendue, valeurCalculée)`
  vérifie l'égalité, ou l'inégalité de deux valeurs ;
  utilisable pour les valeurs primitives (entiers, flottants) et les objets ;
  pour les objets, utilise `equals`
  (attention à l'ordre : valeur attendue, puis valeur calculée, sous peine de messages durs à interpréter)
- `assertSame (valeurAttendue, valeurCalculée)`
  `assertSame (valeurAttendue, valeurCalculée)`
  vérifie l'identité physique des objets, testée avec `==`
- `fail()`
  échoue à tous les coups ;
- il existe des versions où un message (`String`) est passé en argument,
  ce qui permet d'avoir un rapport d'erreur plus détaillé.
  e.g. : `assertEquals("mymethod should return 12", 12, mymethod());`

Langage d'assertion : test d'exceptions

Il est parfois utile de vérifier qu'une exception est bien signalée.
(il faut écrire aussi des tests pour les conditions d'erreur, pas uniquement pour les cas à succès)

test d'exception

```
@Test(expected = NullPointerException.class) public void test() {
    Point p = null;
    p.getX();
}
```

- indiquer la classe de l'exception attendue dans `@Test` avec `expected` (réflexion) ;
- si `NullPointerException` est signalée, le test est réussi ;
- si la méthode se termine sans exception, le test est échoue.

version JUint >= 4.13

```
@Test public void test() {
    assertThrows( NullPointerException.class, () -> { Point p = null; p.getX(); } );
}
```

À partir de JUnit 4.13 :

- utilisation de la méthode `assertThrows`, sur le modèle des autres assertions ;
- le code signalant l'exception est empaqueté dans une lambda : `() -> { ... }` ;
  (plus sur les lambdas dans les prochains cours ; JUnit 5 a également systématiquement recours aux lambdas)
- rend possible le test en séquence de plusieurs assertions.

Cycle d'exécution des tests

exemple

```
public class MyTest {
    private FileReader f;
    @Before public void setUp()   { f = new FileReader("file.txt"); }
    @After public void tearDown() { f.close(); }
    @Test public void test1()     { load("1"); ... }
    @Test public void test2()     { load("2"); ... }
    private void load(String s)   { ... }
}
```

Quand JUnit est lancé sur une classe `MyTest`, alors
pour chaque méthode `test` de la classe, annotée avec `@Test` :

- un objet est créé, avec le constructeur sans argument : `new MyTest()` ;
- les méthodes annotées `@Before` sont appelées (s'il y en a) ;
- la méthode `test` est appelée, sans argument ;
- les méthodes annotées `@After` sont appelées (s'il y en a).

Sortie : nombre de méthodes `@Test` dont toutes les assertions ont été validées.

Notes :

– Une classe de test peut tout à fait avoir des attributs, et des méthodes privées.
– `@Before` et `@After` permettent de factoriser du code commun à tous les tests.
– L'exécution d'une méthode s'arrête à la première assertion qui échoue, et JUnit passe à la méthode suivante.

JUnit avancé

regroupement de tests

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    Test1.class, Test2.class,
    Test3.class, Test4.class
})
public class TestAll {}
```

test paramétrique

```
@RunWith(Parameterized.class)
public class ParamTest {
    @Parameters
    public static Collection<Object[]> data() { ... }
    private int v;
    public ParamTest(int input) { v = input; }
    @Test public void test() { ... }
}
```

Quelques fonctionnalités avancées de JUnit, pour encore plus d'automatisation :

- `@Suite` : combiner l'exécution de plusieurs classes de test existantes en un test ;
- `@Parameters` : exécuter un test sur une liste d'arguments.

Bonnes pratiques :

- une méthode de test (au moins !) par méthode publique de la classe testée, plus des méthodes d'intégration testant des enchaînements de méthodes ;
- tester les cas d'utilisation normale, mais aussi les conditions particulières ;
  (e.g., cas liste vide, cas liste pleine, argument invalide, argument `null`, ...)
- tester les cas où une exception est attendue ;
- une seule assertion par méthode de test.

Annotations

Annotations

`@Test`, `@Override` sont des exemples d'annotation.

Les annotations permettent de développer des extensions du langage, utilisables :

- par le compilateur (e.g., vérification de `@Override`),
- ou par un outil externe (comme Eclipse),
- ou par le programme lui-même, grâce à la réflexion (ou introspection).

Forme générale des annotations :

- `@Annotation`, ou
- `@Annotation(clé1 = valeur1, ..., cléN = valeurN)`

pouvant être placées devant une déclaration de classe, de méthode, d'attribut, ...

Tout nouveau type d'annotation `@A` doit être défini avant utilisation
par un fichier `A.java` définissant une `@interface A` :

Info.java

```
public @interface Info {
    String author();
    String date();
    int version();
}
```

Client.java

```
@Info(author="Gaspard", date="10/12/2015", version=1)
public class Client { ... }
```

Introspection

Rappel : mécanisme de réification du type

- tout objet à une méthode`Class getClass()` ;
- `java.lang.Class` est une représentation réifiée d'une classe Java ;
- toute classe a un attribut statique `class` de type `Class` ;
- la méthode statique `Class.forName(String)` trouve un objet `Class` à partir d'un nom de classe.

Les objets de type `Class` peuvent être parcourus pour :

- lister les attributs, avec : `Field[] getDeclaredFields()`
- lister les méthodes, avec : `Method[] getDeclaredMethods()`

Les classes `Class`, `Field`, `Method` ont alors une méthode :
`Annotation[] getDeclaredAnnotations()`
pour lister les annotations...

Exemple : vérification d'attributs `null`

NonNull.java

```
import java.lang.annotation.*;

// l'annotation est conservée
// dans le .class
@Retention(RetentionPolicy.RUNTIME)
public @interface NonNull { }
```

Exemple.java

```
public class Exemple {
  @NonNull private String name;
  @NonNull private String surname;
  private String nickname;
  ...
}
```

NullChecker.java

```
static boolean check(Object obj) throws ... {
  Class<?> clazz = obj.getClass();
  while (clazz != null) {
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
       field.setAccessible(true);
       Annotation[] annotations =
         field.getDeclaredAnnotations();
       Object val = field.get(obj);
       for (Annotation a : annotations) {
         if (a instanceof NonNull && val == null)
           return false;
       }
    }
    clazz = clazz.getSuperclass();
  }
  return true;
}
```

`NullChecker.check(obj)` vérifie que tous les attributs de `obj` annotés avec `@NonNull` sont effectivement différents de `null`.

En partant de la classe de l'objet (`getClass()`), on parcourt la hiérarchie de classes (`getSuperclass()`) pour énumérer tous les attributs (`getDeclaredAnnotations()`), leur valeur dans l'objet (`get(obj)`) et leurs annotations (`getDeclaredAnnotations()`)...
Voir la documentation de `java.lang.Class` et `java.lang.reflect.Field`.