Programmation Orientée Objet

Bertrand Estellon

Département Informatique et Interactions
Aix-Marseille Université

10 octobre 2012

Fonction de rappel (Callback) – C#

```
public delegate void Callback(int c);

class MaClasse {

  public static void callbackImpl1(int c)
    { Console.WriteLine("{0}",c); }

  public static void callbackImpl2(int c)
    { Console.WriteLine("* {0} *",c); }

  public static void function(int d, Callback callback) {
    /* ... */ callback(d); /* ... */
  }

  static void Main(string[] args) {
    Callback c1 = new Callback(callbackImpl1); function(2,c1);
    Callback c2 = new Callback(callbackImpl2); function(4,c2);
  }
}
```


Les exceptions

Un programme peut être confronté à une condition exceptionnelle (ou exception) durant son exécution.

Une exception est une situation qui empêche l'exécution normale du programme (elle ne doit pas être considérée comme un bug).

Quelques exemples de situations exceptionnelles :

- un fichier nécessaire à l'exécution du programme n'existe pas ;
- division par zéro ;
- débordement dans un tableau ;
- etc.

Mécanisme de gestion des exceptions

Java propose un mécanisme de gestion des exceptions afin de distinguer l'exécution normale du traitement de celles-ci afin d'en faciliter leur gestion.

En Java, une exception est concrétise par une instance d'une classe qui étend la classe `Exception`.

Pour lever (déclencher) une exception, on utilise le mot-clé `throw` :

```
if (problem) throw new MyException("error");
```

Pour capturer une exception, on utilise la syntaxe `try/catch` :

```
try { /* Problème possible */ }
catch (MyException e) { /* traiter l'exception. */ }
```

Définir son exception

Il suffit d'étendre la classe `Exception` (ou une classe qui l'étend) :

```
public class MyException extends Exception {

  private int number;

  public MyException(int number) {
    this.number = number;
  }

  public String getMessage() {
    return "Error "+number;
  }

}
```


La syntaxe try/catch

Pour capturer une exception, on utilise la syntaxe try/catch :

```
public static void test(int i) {
  System.out.print("A ");
  try {
    System.out.println("B ");
    if (i > 12) throw new MyException(i);
    System.out.print("C ");
  } catch (MyException e) { System.out.println(e); }
  System.out.println("D");
}
```

`test(11)` :

```
A B
C D
```

`test(13)` :

```
A B
MyException: Error 13
D
```


Exceptions et signatures des méthodes

Une méthode doit préciser dans sa signature toutes les exceptions qu'elle peut générer et qu'elle n'a pas traitées avec un bloc `try/catch` :

```
public class Test {

  public static void method1(int i) throws MyException {
    method2(i);
  }

  public static void method2(int i) throws MyException {
    if (i>12) throw new MyException(i);
  }

  public static void main(String arg[]) {
    try { method1(i); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```


Pile d'appels

La méthode `printStackTrace` permet d'afficher la pile d'appels :

```
public class Test {
  public static void method1(int i) throws MyException { method2(i); }

  public static void method2(int i) throws MyException {
    if (i>12) throw new MyException(i);
  }

  public static void main(String arg[]) {
    try { method1(i); } catch (MyException e) { e.printStackTrace(); }
  }
}
```

```
MyException: Error 13
        at Test.method2(Test.java:5)
        at Test.method1(Test.java:2)
        at Test.main(Test.java:9)
```

La classe RuntimeException

Une méthode doit indiquer toutes les exceptions qu'elle peut générer sauf si l'exception étend la classe `RuntimeException` Bien évidemment, la classe `RuntimeException` étend `Exception`.

Quelques classes Java qui étendent `RuntimeException` :

- ArithmeticException
- ClassCastException
- IllegalArgumentException
- IndexOutOfBoundsException
- NegativeArraySizeException
- NullPointerException

Notez que ces exceptions s'apparentent le plus souvent à des bugs.

Capturer une exception en fonction de son type

Il est possible de capturer une exception en fonction de son type :

```
public static int diviser(Integer a, Integer b) {
  try { return a/b; }
  catch (ArithmeticException e) {
    e.printStackTrace();
    return Integer.MAX_VALUE;
  } catch (NullPointerException e) {
    e.printStackTrace();
    return 0;
  }
}
```

`diviser(12,0)` :

```
java.lang.ArithmeticException: / by zero
        at Test.diviser(Test.java:17)
        at Test.main(Test.java:28)
```

`diviser(null,12)` :

```
java.lang.NullPointerException
        at Test.diviser(Test.java:17)
        at Test.main(Test.java:28)
```


Le mot-clé finally

Le bloc associé au mot-clé `finally` est toujours exécuté :

```
public static void readFile(String file) {
  try {
    FileReader f = new FileReader(file);
          /* peut déclencher une FileNotFoundException. */
    try {
      int ch = f.read(); /* peut déclencher une IOException. */
      while (ch!=-1) {
        System.out.println(ch);
        ch = f.read(); /* peut déclencher une IOException. */
      }
    } finally { f.close(); /* à faire dans tous les cas. */ }
  } catch (IOException e) { e.printStackTrace(); }
}
```

`FileNotFoundException` étend `IOException` donc elle est capturée.

Le mot-clé finally

il est toujours possible de capturer les exceptions en fonction de leur type :

```
public static void readFile(String file) {
  try {
    FileReader f = new FileReader(file);
          /* peut déclencher une FileNotFoundException. */
    try {
      int ch = f.read(); /* peut déclencher une IOException. */
      while (ch!=-1) {
        System.out.println(ch);
        ch = f.read(); /* peut déclencher une IOException. */
      }
    } finally { /* à faire dans tous les cas. */
          f.close();
    }
  } catch (FileNotFoundException e) {
    System.out.println("File "+file+" not found.");
  } catch (IOException e) { e.printStackTrace(); }
}
```

Exemple

```
public class Stack<T> {
  private Object[] stack;
  private int size;

  public Stack(int capacity) {
    stack = new Object[capacity]; size = 0;
  }

  public void push(T o) throws FullStackException {
    if (taille == stack.length) throw new FullStackException();
    pile[taille] = o; taille++;
  }

  public T pop() throws EmptyStackException {
    if (size == 0) throw new EmptyStackException();
    size--; T t = (T)stack[size]; stack[size]=null;
    return t;
  }
}
```


Exemple

Définition des exceptions :

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

```
public class FullStackException extends StackException {
    public FullStackException() { super("Full stack."); }
}
```

```
public class EmptyStackException extends StackException {
    public EmptyStackException() { super("Empty stack."); }
}
```


Exemple

Exemples d'utilisation :

```
Stack<Integer> stack = new Stack<Integer>(2);
```

```
try {
  stack.push(1);
  stack.push(2);
  stack.push(3);
} catch (StackException e) {
  e.printStackTrace();
}
```

```
FullStackException: Full stack.
        at Stack.push(Stack.java:8)
        at Test.main(Test.java:4)
```

```
try {
  stack.push(1);
  stack.pop();
  stack.pop();
} catch (StackException e) {
  e.printStackTrace();
}
```

```
EmptyStackException: Empty stack.
        at Stack.pop(Stack.java:14)
        at Test.main(Test.java:4)
```


Les énumérations

Il est possible de définir des énumérations :

```
enum Bidule {
    Truc,
    Machin,
    Chose
}
```

Une énumération est une classe avec des éléments prédéfinis et statiques :

```
Bidule b = Bidule.Truc;
/* ... */
if (b == Bidule.Machin) { /* .... */ }
```

Les énumérations

Définition de champs, de méthodes et d'un constructeur :

```
enum Bidule {
  Truc("Truc", 5),
  Machin("Machin", 4),
  Chose("Chose", 8)

  private final double nom;
  private final double prix;

  Bidule(String nom, int prix) {
    this.nom = nom;
    this.prix = prix;
  }

  private double nom()   { return nom; }
  private double prix() { return prix; }
}
```


Les énumérations

Un exemple d'utilisation de l'énumération précédente :

```
public static void main(String[] args) {
  for (Bidule bidule : Bidule.values())
    System.out.printf("Le prix de %s est %d",
                      bidule.nom(),
                      bidule.prix());
}
```


Révision – Les mots réservés

- Gestions des paquets :

  `package`
  `import`
- Définitions des classes, interfaces et énumérations :

  `class`
  `interface`
  `enum`
- Héritage :

  `extends`
  `implements`

Révision – Les classes

```
class MyClass {

  int field1;
  String field2;

  MyClass(int param1) {
    field1 = param1;
  }

  int method(int param1, int param2) {
    return param1 + param2 + field1;
  }

}
```

Révision – Schéma d'une classe

Un exemple de schéma représentant une classe :

MaClasse
+ field1 : List<Integer> + field2 : String
+ MaClasse(s : String) + method1(a : int, b : int) + method2(b : int) : String

Les règles d'écriture des champs et des méthodes :

- **Champ** → `nom ':' type`
- **Méthode** → `nom '(' arguments ')' ':' type`
- **Arguments** → `argument ( ',' argument )*`
- **Argument** → `nom ':' type`

Révision – Interfaces

```
interface MyInterface {
  /**
   * Description du comportement de la méthode.
   * @param p ...
   */
  void method1(int p);

  /**
   * Description du comportement de la méthode.
   * @param p ...
   * @return ....
   */
  int method2(int p);
}
```


Révision – Implémentation d'une interface

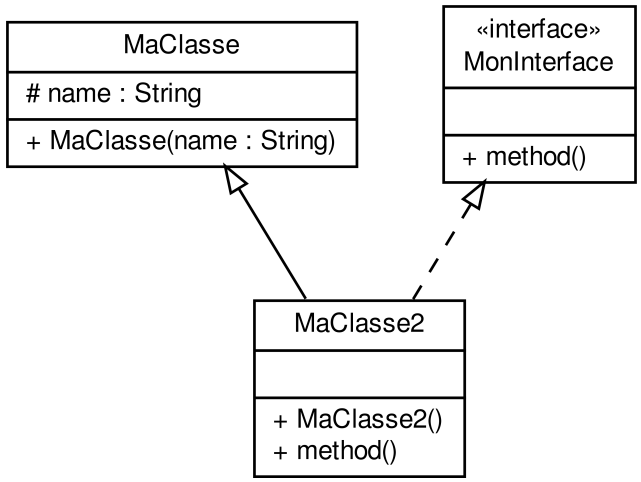
```
class MyClass implements MyInterface {

  String name;

  MyClass(String name) {
    this.name = name;
  }

  void method1(int p1) { System.out.println(p1); }

  int method2(int p1, int p2) { return p1+p2; }
}
```


Révision – Extension d'une classe

```
class MyOtherClass extends MyClass {

 int number;

 MyOtherClass(String name, int number) {
  super(name);
  this.number = number;
 }

 void method1(int p1)
       { System.out.println(name+" : "+p1); }

 int method3(int p) { return p+number; }

}
```

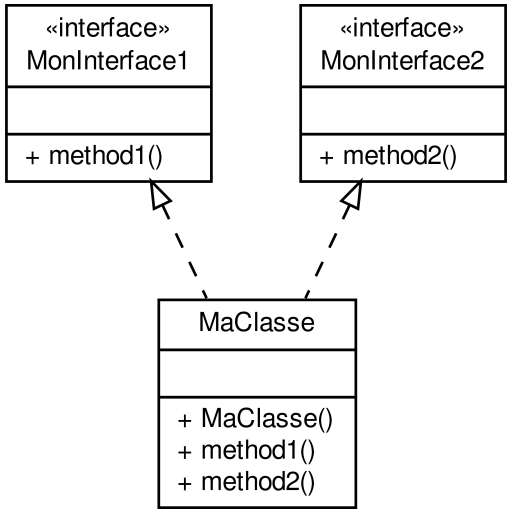
Révision – Interfaces, extensions et implémentations

Une interface, une extension et une implémentation :



Révision – Implémentation multiples

Une implémentation multiple :



Révision – Polymorphisme

Une interface et deux classes qui l'implémentent :

```
interface I { void method(); }

class A implements I {
  void method() { System.out.println("A"); }
}

class B implements I {
  void method() { System.out.println("B"); }
}
```

Laquelle des deux méthodes est appelée ?

```
int test(boolean choix)
  I i; if (choix) i = new A(); else i = new B();
  i.method();
}
```


Révision – Polymorphisme

Une classe qui étend une classe :

```
class A {
 void method() { System.out.println("A"); }
}

class B extends A {
 void method() { System.out.println("B"); }
}
```

Laquelle des deux méthodes est appelée ?

```
int test(boolean choix)
 A a; if (choix) a = new A(); else i = new B();
 a.method();
}
```

Révision – Les mots réservés

Modificateurs pour les membres et les variables :

`public/protected/private`
`static`
`final`
`abstract`
`synchronized`
`throws`

Révision – modificateur public

Par défaut, une classe ou une méthode est non-publique : elle n'est accessible que depuis les classes du même paquet.

Une classe ou un membre public est accessible de n'importe où.

Pour rendre une classe ou un membre public :

```
public class ClassePublique {
 public int proprietePublique;
 public void methodePublique() { }
}
```

Si un fichier contient une classe publique, le nom du fichier doit être formé du nom de la classe suivi de ".java".

Un fichier ne peut contenir qu'une seule classe publique.

Révision – modificateur private

Un membre privé n'est accessible que par les méthodes de la classe qui le contient.

Pour rendre un membre privé, on utilise le modificateur `private` :

```
public class ClassePublique {
 private int proprietePrivee;
 private void methodePrivee() { }
}
```

Encapsulation : tout ce qui participe à l'implémentation des services doit être privé afin de permettre la modification de l'implémentation des services sans risquer d'impacter les autres classes.

Révision – Modification protected

Un membre `protected` n'est accessible que par les méthodes des classes du paquet et par les classes qui l'étendent.

Le modificateur `protected` permet de protéger un membre :

```
public class ClassePublique {
 protected int proprieteProtegee;
 protected void methodeProtegee() { }
}
```

	Classe	Paquet	Extension	Extérieur
Private	✓			
Default	✓	✓		
Protected	✓	✓	✓	
Public	✓	✓	✓	✓

Révision – Données et méthodes statiques

Les méthodes et des données statiques sont associées à la classe et non aux instances de la classe :

```
class Counter {
 private static int counter = 0;
 static void count() { counter++; }
 static int value() { return counter; }
}
```

Un exemple d'utilisation de la classe précédente :

```
Counter.count();
Counter.count();
Counter.count();
System.out.println(Counter.value());
```


Révision – Classes abstraites

```
abstract class A {
 abstract int getNumber();
 void printNumber() { System.out.println(getNumber()); };
}

class B extends A {
 int getNumber() { return 5; }
}

class C extends A {
 int getNumber() { return 10; }
}
```


Révision – Classes abstraites

Les deux classes B et C héritent de la méthode `getNumber()` :

```
B b = new B();
C c = new C();
b.getNumber();
c.getNumber();
```

Classes abstraites et polymorphisme :

```
A a;
if (test) a = new B(); else a = new C();
a.printNumber();
```


Révision – Les mots réservés dans le code

```
if/else
for
do/while
switch/case/default
continue/break
try/catch/finally
throw
return
new
null
false
true
this
super
```

Révision – this

```
class MyClass {
  private String name;
  private int number;

  public MyClass(String name, int number) {
    this.name = name;
    this.number = number;
  }

  int addNumber(int number) {
    return number + this.number;
  }
}
```


Révision – super

```
class MyClass {
  private String name;
  public Truc(String name) { this.name = name; }
  public String toString() { return name; }
}

class MyOtherClass extends MyClass {
  private int number;

  public Machin(String name, int number) {
    super(name); this.number = number;
  }
  public String toString() {
    return super.toString() + " "+number;
  }
}
```


Révision – Les mots réservés et les types simples

```
boolean
byte
char
short
int
long
double
float
void
```


Révision – Les types primitifs

byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flotant	32 bits		0.0
double	flotant	64 bits		0.0
char	caractère	16 bits	caractères Unicode	\u0000
boolean	boolean	1 bit	false ou true	false

```
int a = 12;
double b = 13.5;
```

Révision – Classes génériques

Définition d'une classe générique :

```
class MyClass<T> {

  void method1(T e) { ... }
  T method2() { ... }

}
```


Révision – Les autres mots réservés

```
assert
goto
native
assert
strictfp
volatile
instanceof
transient
```


Révision – Programme bien conçu

Un programme est "bien conçu" s'il permet de :

- Absorber les changements avec un minimum d'effort
- Implémenter les nouvelles fonctionnalités sans toucher aux anciennes
- Modifier les fonctionnalités existantes en modifiant localement le code

Objectifs :

- Limiter les modules impactés
  - ⇒ Simplifier les tests unitaires
  - ⇒ Rester conforme à la partie des spécifications qui n'ont pas changé
  - ⇒ Facilité l'intégration
- Gagner du temps

Remarque : Le développement d'une application est une suite d'évolutions.

Révision – Les cinq principes (pour créer du code) SOLID

- Single Responsibility Principle (SRP) :
  Une classe ne doit avoir qu'une seule responsabilité
- Open/Closed Principle (OCP) :
  Programme ouvert pour l'extension, fermé à la modification
- Liskov Substitution Principle (LSP) :
  Les sous-types doivent être substituables par leurs types de base
- Interface Segregation Principle (ISP) :
  Éviter les interfaces qui contiennent beaucoup de méthodes
- Dependency Inversion Principle (DIP) :
  Les modules d'un programme doivent être indépendants
  Les modules doivent dépendre d'abstractions

Révision – Patrons de conception (Design patterns)

- Les patrons de conception décrivent des solutions standards pour répondre aux problèmes rencontrés lors de la conception orientée objet
- Ils tendent à respecter les 5 principes SOLID
- Ils sont le plus souvent indépendants du langage de programmation
- Ils ont été formalisés dans le livre du "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – 1995)
- "Les patrons offrent la possibilité de capitaliser un savoir précieux né du savoir-faire d'experts" (Buschmann – 1996)
- Les anti-patrons (ou antipatterns) sont des erreurs courantes de conception.

Les utilisations de Java

- Interfaces graphiques (Swing, AWT, SWT...)
- Applications réseaux (RMI...)
- Site Web (JFace, JSP, DOM, Servlets, GWT...)
- Gestion des formats de données (XML, Excels...)
- Bases de données (JDBC, JDO, Hibernate...)
- Gestion des tests unitaires (JUnit)
- Application sur téléphones mobiles (J2ME, Android...)
- et bien d'autres utilisations

Pour utiliser ces technologies, les concepts de POO sont indispensables.