Surcharge et Exceptions

Arnaud Labourel arnaud.labourel@univ-amu.fr

13 octobre 2021

Section 1

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

Dans une classe, plusieurs méthodes peuvent avoir le même nom.

C'est ce qu'on appelle la surcharge de méthode.

Il est par contre nécessaire que la séquence dans l'ordre des types des arguments soit différente pour chaque méthode ayant le même nom.

La méthode est choisie par le compilateur de la façon suivante :

- Le nombre de paramètres doit correspondre
- Les affectations des paramètres doivent être valides
- Parmi ces méthodes, le compilateur choisit la plus spécialisée, c'est-à-dire celle entraînant le moins de changement de types (passage à une super-classe ou promotion pour les types primitifs)

Exemple de surcharge de méthode

```
public class DataArtist {

    public void draw(String s) {
        /* ... */
    }
    public void draw(int i) {
        /* ... */
    }
    public void draw(double f) {
        /* ... */
    }
    public void draw(int i, double f) {
        /* ... */
    }
}
```

Exemple de surcharge de méthode

```
class Adder {
  public static int add(int intVal1, int intVal2) {
    System.out.println("integer");
    return intVal1+intVal2;
  }

  public static double add(double doubleVal1,
                           double doubleVal2) {
    System.out.println("double");
    return doubleVal1+doubleVal2;
  }
}
```

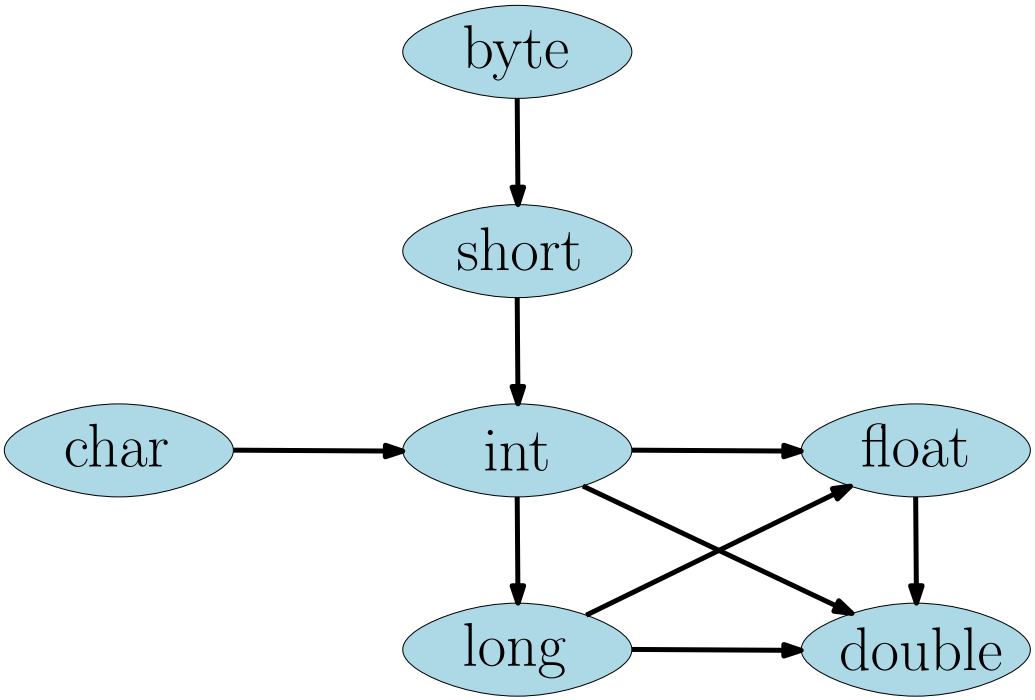
Exemple de surcharge de méthode

```
int intValue = 1;
double doubleValue = 2.2;
double result1 = Adder.add(doubleValue, doubleValue);
// → double

int result3 = Adder.add(intValue, intValue);
// → int
```

Promotion pour les types primitifs

Si les types des arguments ne correspondent pas aux types des paramètres, les types des arguments sont promus en suivant les flèches :




```
double result2 = Adder.add(intValue, doubleValue);
// → double
```

Choix de la méthode à la compilation

Pour les instances de classe, c'est le type du conteneur à la compilation qui compte (et pas le type réel).

```
class Printer {
  static void print(Object object) {
    System.out.println("Object : "+object);
  }
  static void print(String string) {
    System.out.println("String : "+string);
  }
}
String string = "message";
Object object = string;
Printer.print(string); // → String : message
Printer.print(object); // → Object : message
```

Choix de la méthode : coût de conversion

Lorsque plusieurs méthodes ont une signature compatible (via des transtypages), c'est la méthode dont le coût de conversion (mesuré par les distances dans l'arbre d'héritage) est la plus faible qui est choisie.

Exemple

- une classe `Surgeon` qui étends `Employee` elle-même étendant `Person`
- deux méthodes `setAppointment(Employee e1, Employee e2)` et `setAppointment(Employee e, Person p)`.

→ si on appelle `setAppointment(s1, s2)` avec deux `Surgeon` alors c'est `setAppointment(Employee e1, Employee e2)` qui est appelé car sont coût de conversion est de 2.

Erreur possible

deux méthodes avec le même coût ⇒ erreur de compilation

Plusieurs constructeurs

C'est exactement les mêmes règles qui s'appliquent pour les constructeurs d'une classe.

```
public class Point{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0); // This appelle le constructeur de l
    }
    public Point(Point p) {
        this(p.x, p.y);
    }
}
```

Section 2

Exceptions

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 toujours être considérée comme un bug). Quelques exemples de situations exceptionnelles :

- un fichier nécessaire à l'exécution du programme n'existe pas,
- une division par zéro,
- un débordement dans un tableau,
- un besoin de se connecter à un serveur et celui-ci est injoignable,
- un dépilement d'une pile vide,
- ...

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 concrétisée 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 les mots-clés `try` et `catch` :

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

Exemple d'exceptions existantes en java

- `ArithmeticException` : opération arithmétique impossible comme la division par 0.
- `IndexOutOfBoundsException` : dépassement d'indice dans un tableau, un vecteur, ...
- `NullPointerException` : accès à un attribut/méthodes/case pour les tableaux d'une référence valant `null`, argument `null` alors que ce n'est pas autorisé.
- `FileNotFoundException` : échec de l'ouverture d'un fichier à partir d'un chemin.
- `IllegalArgumentException` : argument incorrect (en dehors des valeurs autorisées) lors de l'appel d'une méthode.
- `NoSuchElementException` : `next` alors que l'itération est finie, dépilement d'une pile vide, ...
- ...

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 value) {
  System.out.print("A ");
  try {
    System.out.println("B ");
    if (value > 12) throw new MyException(value);
    System.out.print("C ");
  } catch (MyException e) { System.out.println(e); }
  System.out.println("D");
}
```

`test(11)`	`test(13)`
A B	A B
C D	MyException: Error 13
	D

Exceptions et signatures des méthodes

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

```
public static void testValue(int value)
                    throws MyException {
  if (value>12) throw new MyException(value);
}
public static void runTestValue(int value)
                    throws MyException {
  testValue(value);
}
```

La méthode `testValue` peut lever une exception de type `MyException`.

`runTestValue` doit indiquer qu'elle peut lever une exception car elle ne gère pas l'exception provoquée par l'appel `testValue(value)`.

Gestions des exceptions : règle

La méthode `runTestValue` peut lever une exception (de type `MyException`).

Lorsqu'on fait un appel à la méthode `runTestValue`, il est vérifié à la compilation que l'une des deux propriétés suivante est vraie :

- la méthode appelant `runTestValue` est indiquée comme pouvant lever l'exception `MyException` (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un loc `try/catch`.

Si aucune des deux propriétés est vérifiée alors il y a une erreur à la compilation.

```
Error:(YY, XX) java: unreported exception MyException;
      must be caught or declared to be thrown
```

Exceptions et signatures des méthodes

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

On doit donc écrire :

```
public class Main {
  public static void main(String[] args)  {
    try { Test.runTestValue(13); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```

Techniquement la méthode `main` pourrait indiquer qu'elle génère l'exception `MyException` mais cela n'aurait pas beaucoup de sens car cela voudrait dire qu'on ne gère pas vraiment l'exception.

Méthode printStackTrace

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

```
public class Main {
  public static void main(String[] args)  {
    try { Test.runTestValue(13); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```

```
MyException: Error 13
    at Test.testValue(Test.java:23)
    at Test.runTestValue(Test.java:20)
    at Main.main(Main.java:6)
```

La classe `RuntimeException`

Java définit une classe étendant `Exception` nommée `RuntimeException`.

Les `RuntimeException` correspondent généralement à des bugs pouvant arriver très fréquemment (références non initialisées, divisions par zéro, mauvaises valeur d'arguments, ...) qu'on n'a pas obligation de gérer :

- `ArithmeticException`
- `ClassCastException`
- `IllegalArgumentException`
- `IndexOutOfBoundsException`
- `NegativeArraySizeException`
- `NullPointerException`
- `NoSuchElementException`

Règle générale : ajout cas `RuntimeException`

Lorsqu'on fait un appel à une méthode `canThrow` pouvant lever une exception `MyException` qui n'étend pas `RuntimeException`, il est vérifié à la compilation que l'une des deux propriétés suivantes est vraie :

- la méthode appelant `canThrow` dans son code est indiquée comme pouvant lever une exception de type `MyException` ou une de ces super-classes (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un bloc `try/catch`.

$\Rightarrow$ On n'a pas obligation de gérer les `RuntimeException` mais on peut le faire si on le souhaite.

Capturer une exception en fonction de son type

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

```
divide(12,0) :
java.lang.ArithmeticException: / by zero
  at Test.diviser(Test.java:17)
  at Test.main(Test.java:28)
```

Le mot-clé finally

On rajouter un bloc finally après des blocs try. Le bloc associé au mot-clé finally est toujours exécuté :

```
public static void readFile(String fileName) {
  try {
    FileReader fileReader = new FileReader(fileName);
      /* peut déclencher une FileNotFoundException. */
    try {
      int character = fileReader.read();// IOException ?
      while (character!=-1) {
        System.out.println(character);
        character = fileReader.read(); // IOException ?
      }
    } finally { fileReader.close();/*dans tous les cas*/
  } catch (IOException exception) {
    exception.printStackTrace();
```

Gestion de différents types d'exceptions

```
try { FileReader fileReader = new FileReader(fileName);
  try {
    int character = fileReader.read(); // IOException ?
    while (character!=-1) {
      System.out.println(character);
      character = fileReader.read(); // IOException ?
    }
  } finally { /* à faire dans tous les cas. */
    fileReader.close();
  }
} catch (FileNotFoundException exception) {
  System.out.println("File "+fileName+" not found.");
} catch (IOException exception) {
  exception.printStackTrace();
}
```

Exceptions pour des piles (1/2)

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

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

Exceptions pour des piles (2/2)

```
public class Stack<T> {
  public void push(T object) throws FullStackException {
    if (size == stack.length)
      throw new FullStackException();
    stack[size] = object;
    size++;
  }
  public T pop() throws EmptyStackException {
    if (size == 0) throw new EmptyStackException();
    size--;
    T object = (T)stack[size];
    stack[size]=null;
    return object;
  }
}
```

Définition des exceptions pour les piles

```
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 d'utilisation (1/2)

```
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:13)
    at Main.main(Main.java:10)
```

Exemple d'utilisation (2/2)

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

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

```
EmptyStackException: Empty stack.
    at Stack.pop(Stack.java:18)
    at Main.main(Main.java:10)
```