Programmation 2 : sixième cours

Arnaud Labourel arnaud.labourel@univ-amu.fr

5 ou 7 novembre 2018

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

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

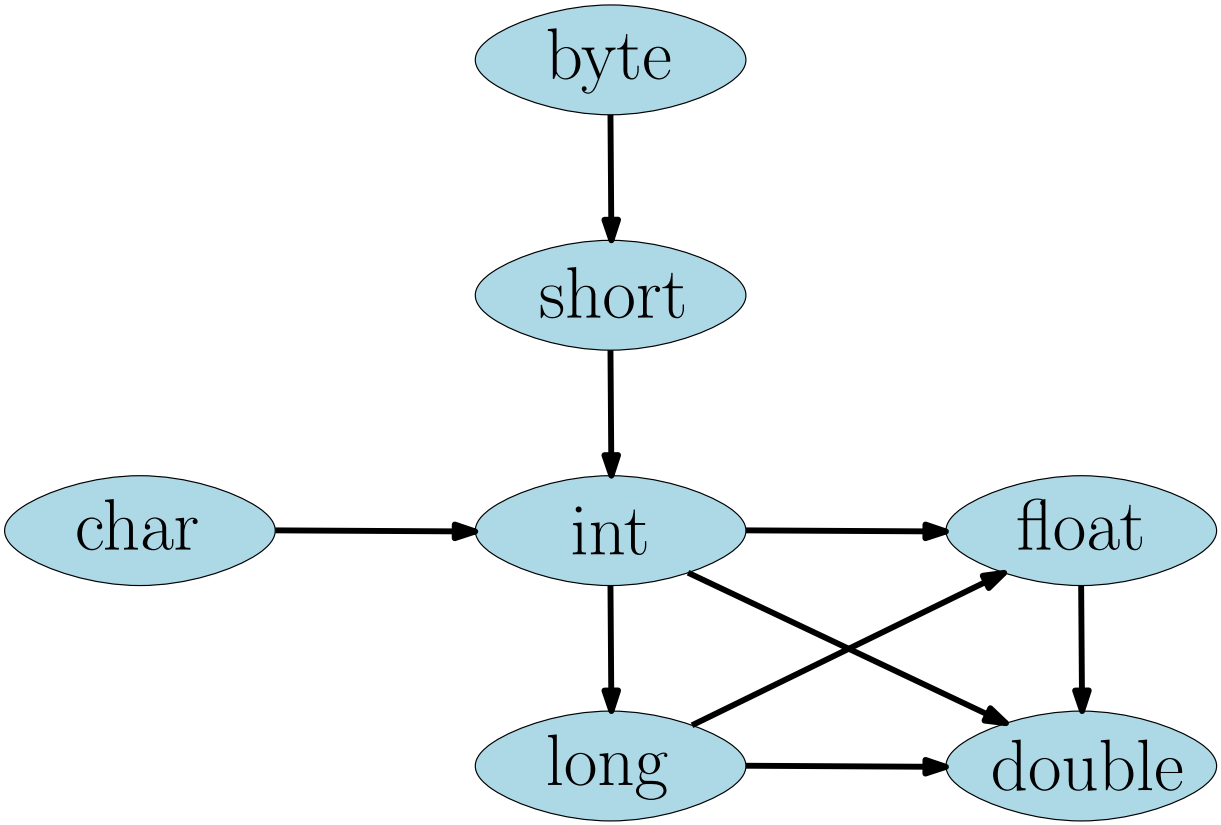
Exemple de surcharge de méthode

```
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
```

Exemple de surcharge de méthode

Pour les instances de classe, c'est le type du conteneur à la compilation qui compte.

```
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
```

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);
    }
    public Point(Point p) {
        this(p.x, p.y);
    }
}
```

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 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,
- l'application a besoin de se connecter à un serveur et celui-ci est injoignable,
- dépiler un objet d'un 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`.
- `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

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`

Une méthode doit indiquer toutes les exceptions qu'elle peut lever 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`
- `NoSuchElementException`

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

Description de la classe `RuntimeException` dans la documentation Java

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

`RuntimeException` and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Règle générale

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`.

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)
```

Les énumérations

Énumérations en Java

En programmation, une *énumération* est un type spécial qui permet de définir des variables pouvant prendre des valeurs parmi un ensemble prédéfini de constantes.

Il est possible de définir des énumérations en Java grace au mot-clé `enum`

```
enum Suit{
  SPADES,
  HEARTS,
  DIAMONDS,
  CLUBS;
}
```

Énumérations en Java

```
enum Suit {
  SPADES, HEARTS, DIAMONDS, CLUBS;
}
```

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

On peut donc tester directement l'égalité avec l'opérateur égal car il n'y qu'un objet et donc qu'une référence qui correspond à chaque valeur possible.

```
Suit suit = Suit.SPADES;
/* ... */
if (suit == Suit.SPADES)
{
  /* .... */
}
```

Définition de champs, de méthodes et d'un constructeur

```
public enum Suit {
  SPADES("Pique", "Pi"), HEARTS("Coeur", "Co"),
  DIAMONDS("Carreau", "Ca"), CLUBS("Trèfle", "Tr");

  private final String frenchName;
  private final String frenchSymbol;

  Suit(String name, String symbol) {
    this.frenchName = name;
    this.frenchSymbol = symbol;
  }
  public String frenchName() { return frenchName; }
  public String frenchSymbol() { return frenchSymbol; }
}
```

Classe `Enum`

Toutes les énumération de java étendent la classe `Enum`.

Quelques méthodes utiles :

- `String name()` : retourne le nom de la constante tel que déclaré dans l'`enum`.
- `int ordinal()` : retourne la position de la constante dans la déclaration de l'`enum` (commençant par 0).

Tout `enum` définit aussi une méthode statique `values()` renvoyant un tableau contenant les constantes dans l'ordre de leurs déclarations.

Exemple d'utilisation d'énumération (1/2)

```
public static void main(String[] args) {
  Suit[] values = Suit.values();
  for (Suit suit : values)
  System.out.printf("Le symbole de %s est %s.\n",
          suit.frenchName(), suit.frenchSymbol());
}
```

Affichage :

```
Le symbole de Pique est Pi.
Le symbole de Coeur est Co.
Le symbole de Carreau est Ca.
Le symbole de Trèfle est Tr.
```

Exemple d'utilisation d'énumération (2/2)

```
public static void main(String[] args) {
  for (Suit suit : Suit.values())
  System.out.printf("The position of %s is %d.\n",
                    suit.name(), suit.ordinal());
}
```

Affichage :

```
The position of SPADES is 0
The position of HEARTS is 1
The position of DIAMONDS is 2
The position of CLUBS is 3
```

Mot-clé `switch` et `enum`

Supossons qu'on est une `enum` pour les jour de la semaine :

```
enum Day
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY;
}
```

Mot-clé switch et enum

```
public void dayIsLike() {
  switch (day) {
    case MONDAY:
      System.out.println("Mondays are bad.");
      break;
    case FRIDAY:
      System.out.println("Fridays are better.");
      break;
    case SATURDAY:
    case SUNDAY:
      System.out.println("Weekends are best.");
      break;
    default:
      System.out.println("Midweek days are so-so.");
      break;
  }
```

Mot-clé `final`

Première utilisation du mot-clé `final`

Mot-clé `final` dans la déclaration d'un attribut : interdit la modification de l'attribut après la construction de l'objet.

Exemple :

```
public class Integer {
  private final int value;
  public Integer(int value) {
    this.value = value;
  }
}
```

- Un attribut `final` doit être initialisé après la construction de l'instance
- La valeur de l'attribut ne peut plus être modifiée ensuite

Deuxième utilisation du mot-clé `final`

Mot-clé `final` dans la déclaration d'une variable : interdit la modification de la variable après la première affectation.

```
public class Stack<T> { /* ... */
  public T pop() {
    final T top = array[size-1];
    array[size-1] = null;
    size--;
    return top;
  }
}

public final class Math {
  public static final double PI = 3.14159265358979323846;
}
```

Troisième utilisation du mot-clé `final`

Mot-clé `final` dans la déclaration d'une méthode : interdire la redéfinition d'une méthode dans une sous-classe

```
public class Integer {
/* ... */
  final public Integer add(Integer val) {
    return new Integer(this.value + val.value);
  }
}
```

- Une classe étendant `Integer` ne peut pas redéfinir (donner une nouvelle implémentation) la méthode `add`

Quatrième utilisation du mot-clé `final`

Mot-clé `final` dans la déclaration d'une classe : interdit l'extension de la classe

```
final public class Integer {
  /* ... */
}
```

- Il devient impossible de créer une classe étendant `Integer`
- On le fait souvent pour des raisons de sécurité et d'efficacité
- De nombreuses classes de la bibliothèque standard de Java sont `final` comme `Math`, `String` et `System`