S4IC020 - Programmation Orientée Objet (POO)

Cours 6 : les types génériques et les paquetages

Florian Boudin

Département d'informatique, Université de Nantes

2011–12

Plan

Introduction (1)

- Les *bugs* sont inévitables dans des applications complexes
- Certaines erreurs sont plus faciles à corriger
  - Erreurs de compilation $\Rightarrow$ message du compilateur
  - Erreurs dans l'exécution $\Rightarrow$?
- Les types génériques permettent d'accroitre la stabilité du code en rendant les erreurs dans l'exécution détectable à la compilation

La classe Box

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

- `Box` accepte et retourne des objets, il est possible de lui passer n'importe quoi (excepté des types primitifs)
- Pour restreindre le type d'objet, la seule option consiste à le spécifier dans la documentation ⇒ `BoxDemo1`

La classe BoxDemo1

```
public class BoxDemo1 {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();
        integerBox.add(new Integer(10));
        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```

- Le programme créé un objet `Integer` et le passe à `add`, puis il assigne ce même objet à `someInteger` par la valeur de retour de `get`
- Le compilateur ignore que le transtypage est correct et ne fera rien pour prévenir d'une éventuelle erreur de programmation

La classe BoxDemo2

```
public class BoxDemo2 {
    public static void main(String[] args) {
        // ONLY place Integer objects into this box!
        Box integerBox = new Box();
        // Imagine this is one part of a large
        // application modified by one programmer.
        // Note how the type is now String.
        integerBox.add("10");
        // ... and this is another, perhaps written
        // by a different programmer
        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);
    }
}
```

```
Exception in thread "main"
  java.lang.ClassCastException:
    java.lang.String cannot be cast to java.lang.Integer
      at BoxDemo2.main(BoxDemo2.java:6)
```

Plan

Mise à jour de la classe Box

```java
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

```java
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

- Introduire le type variable `T` utilisé dans la classe
  - `public class Box` → `public class Box<T>`
  - Remplacement des occurrences de `Object` par `T`
  - Considérer T comme un type spécial de variable
  - Possibilité d'utiliser plusieurs paramètres de type, e.g. `Box<T,U>`

Les classes génériques (1)

- Pour référencer la classe générique, utilisez une invocation de type générique, en remplaçant le `T` par un type concret

```
Box<Integer> integerBox;
```

- Une invocation de type générique peut être considérée comme une invocation à une méthode ordinaire
  - Passer un paramètre à une méthode → un type à une classe
  - Une invocation de type générique = type paramétré
- Pour instancier la classe

```
integerBox = new Box<Integer>();

// Les deux etapes en meme temps
Box<Integer> integerBox = new Box<Integer>();
```

La classe BoxDemo3

```
public class BoxDemo3 {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        // no cast!
        Integer someInteger = integerBox.get();
        System.out.println(someInteger);
    }
}
```

- Si vous essayez d'ajouter un objet de type différent

```
BoxDemo3.java:5: add(java.lang.Integer) in
    Box<java.lang.Integer>
        cannot be applied to (java.lang.String)
            integerBox.add("10");
                           ^
1 error
```

Les classes génériques (2)

- A partir de Java 7, l'argument de type peut être omis (opérateur diamant <>) tant que le compilateur peut inférer le type en fonction du contexte, e.g.

```
Box<Integer> integerBox = new Box<>();
// est identique a
Box<Integer> integerBox = new Box<Integer>();
```

- Conventions de nommage : les paramètres de type sont des lettres en majuscule
  - E - Element (utilisé dans les collections Java)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S, U, V, etc. - 2^{nd}, 3^{ieme}, 4^{ieme} types

Plan

Méthodes et constructeurs génériques

- Les paramètres de type peuvent également être spécifiés dans les méthodes et les constructeurs
- La portée du type de paramètre est limitée à la méthode ou au constructeur dans lequel il est déclaré
- Exemple : ajout d'une méthode générique à la classe `Box`, nommée `inspect` qui définit un type de paramètre `U`
  - Accepte un objet et affiche son type
  - Signature : `public <U> void inspect(U u)`

Box avec méthode générique (1)

```
public class Box<T> {
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

Box avec méthode générique (1)

```java
public class Box<T> {
    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}

// T: java.lang.Integer
// U: java.lang.String
```

Box avec méthode générique (2)

- Une utilisation plus réaliste des méthodes génériques peut être la suivante, qui définit une méthode statique qui emmagasine des références à un même item dans plusieurs boites

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes){
    for (Box<U> box : boxes) {
        box.add(u);
    }
}
```

- Exemple d'utilisation de cette méthode

```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;

// compiler infers that U is Crayon
// Box.<Crayon>fillBoxes(red, crayonBoxes);
Box.fillBoxes(red, crayonBoxes);
```

Plan

Sous-typage (1)

- Il est possible d'assigner un objet d'un type à un objet d'un autre type si ils sont compatibles

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger; // OK
```

- Il s'agit de la relation "is a" (est un)

```
public void someMethod(Number n){...}

someMethod(new Integer(10));  // OK
someMethod(new Double(10.1)); // OK
```

Sous-typage (2)

- Le même principe est applicable aux types génériques

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));  // OK
box.add(new Double(10.1)); // OK
```

- Considérons la méthode suivante

```
public void boxTest(Box<Number> n){...}
```

- Quel type d'argument la méthode accepte-elle ?
  - Argument de type `Box<Number>`
- Peut-on lui passer un `Box<Integer>` ou `Box<Double>` ?

Sous-typage (2)

- Le même principe est applicable aux types génériques

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));  // OK
box.add(new Double(10.1)); // OK
```

- Considérons la méthode suivante

```
public void boxTest(Box<Number> n){...}
```

- Quel type d'argument la méthode accepte-elle ?
  - Argument de type `Box<Number>`
- Peut-on lui passer un `Box<Integer>` ou `Box<Double>` ?
  - De manière surprenante, la réponse est **non**

Sous-typage (3)

- Comprendre avec des exemples plus tangibles

```
// A cage is a collection of things,
// with bars to keep them in.
interface Cage<E> extends Collection<E>;

// A lion is a kind of animal,
// so Lion would be a subtype of Animal
interface Lion extends Animal {}
Lion king = ...;

// Where we need some animal, we're free
// to provide a lion
Animal a = king;

// A lion can of course be put into a lion cage
Cage<Lion> lionCage = ...;
lionCage.add(king);
```

Sous-typage (4)

```
// and a butterfly into a butterfly cage
interface Butterfly extends Animal {}
Butterfly monarch = ...;
Cage<Butterfly> butterflyCage = ...;
butterflyCage.add(monarch);

// But what about an "all-animal cage" ?
Cage<Animal> animalCage = ...;

// This is a cage designed to hold all kinds of
// animals, mixed together. It must have bars
// strong enough to hold in the lions, and spaced
// closely enough to hold in the butterflies.
animalCage.add(king);
animalCage.add(monarch);
```

Sous-typage (5)

- Puisqu'un lion est un animal, est-ce qu'une cage à lion est une sorte de cage à animaux ?
  - Est ce que `Cage<Lion>` est un sous-type de `Cage<Animal>` ?
  - La réponse est **non**, une cage à lion ne peut contenir de papillons et inversement

```
animalCage = lionCage;      // compile-time error
animalCage = butterflyCage; // compile-time error
```

- Sans les types génériques, les animaux pourraient être mis dans les mauvaises cages et s'échapper

Plan

Questions

```
public class AnimalHouse<E> {
    private E animal;
    public void setAnimal(E x) {
        animal = x;
    }
    public E getAnimal() {
        return animal;
    }
}
public class Animal{...}
public class Cat extends Animal {...}
public class Dog extends Animal {...}
```

- Quel est le comportement des morceaux de code suivants :

```
AnimalHouse<Animal> house = new AnimalHouse<Cat>(); // 1

AnimalHouse<Cat> house = new AnimalHouse<Animal>(); // 2

AnimalHouse house = new AnimalHouse();              // 4
house.setAnimal(new Dog());
```

Réponses

```
AnimalHouse<Animal> house = new AnimalHouse<Cat>(); // 1
// Fails to compile.
// AnimalHouse<Cat> and AnimalHouse<Animal> are not
// compatible types, even if Cat is a subtype of Animal

AnimalHouse<Cat> house = new AnimalHouse<Animal>(); // 2
// Fails to compile.
// AnimalHouse<Cat> and AnimalHouse<Animal> are not
// compatible types, even if Cat is a subtype of Animal

AnimalHouse house = new AnimalHouse();                  // 4
house.setAnimal(new Dog());
// Compiles with a warning.
// The compiler doesn't know what type house contains.
// It will accept the code, but warn that there might be
// a problem when setting the animal to an instance of Dog
```

Plan

Création et utilisation de paquetages

Définition

Un paquetage est un regroupement des *types* similaires qui fournit la protection d'accès et la gestion de l'espace de nom. Les *types* réfèrent aux classes, interfaces, énumérations et annotations.

- Les programmeurs groupent les *types* en paquetages pour :
  - les rendre plus facile d'accès et d'utilisation
  - éviter les conflits de nommage
  - contrôler l'accès aux membres
- Les types qui font partie de la plateforme Java sont des membres de différents paquetages qui regroupent les classes par fonction : les classes fondamentales sont dans `java.lang`, les classes pour lire et écrire (entrée et sortie) sont dans `java.io`, etc.

Exemple

- Groupez les classes qui représentent des objets graphiques

```
//in the Draggable.java file
public interface Draggable {}

//in the Graphic.java file
public abstract class Graphic {}

//in the Circle.java file
public class Circle extends Graphic implements Draggable {}

//in the Rectangle.java file
public class Rectangle extends Graphic implements Draggable {}

//in the Point.java file
public class Point extends Graphic implements Draggable {}

//in the Line.java file
public class Line extends Graphic implements Draggable {}
```

Création de paquetages

- Créer un paquetage : ajouter l'instruction `package nom;`

```
//in the Draggable.java file
package graphics;
public interface Draggable {}

//in the Graphic.java file
package graphics;
public abstract class Graphic {}
...
```

- Convention de nommage
  - Les noms de paquetage sont écrits en minuscules
  - Compagnies/programmeurs utilisent le nom de domaine inversé
    e.g. `com.example.region.mypackage`

Importation de paquetages

- L'instruction `import` est utilisée pour importer un membre ou l'intégralité d'un paquetage

```
import graphics.Rectangle;
// Importer la classe Rectangle de graphics
Rectangle myRectangle = new Rectangle();

import graphics.*;
// Importer les classes du paquetage
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();

import static java.lang.Math.PI;
// Importer les membres statiques
double r = cos(PI * theta);
```