Les classes internes

La plupart du temps, une classe en Java est déclarée dans un fichier portant le même nom que la classe avec l'extension *.java*. Cependant, il est également possible de déclarer des classes dans une classe. On parle alors de classes internes (*inner classes*). Cela est également possible, dans une certaine limite, pour les interfaces et les énumérations.

La déclaration des classes internes peut se faire dans l'ordre que l'on souhaite à l'intérieur du bloc de déclaration de la classe englobante. Les classes internes peuvent être ou non déclarées **static**. Ces deux cas correspondent à deux usages particuliers des classes internes.

```
package com.cgi.udev;

public class ClasseEnglobante {

  public static class ClasseInterneStatic {
  }

  public class ClasseInterne {
  }

}
```

Les classes internes static

Les classes internes déclarées **static** sont des classes pour lesquelles l'espace de noms est celui de la classe englobante.

```
package com.cgi.udev;

public class ClasseEnglobante {

  public static class ClasseInterne {
  }

}
```

Pour une classe interne **static** :

- Son nom complet inclus le nom de la classe englobante (qui agit comme un package). Pour la classe ci-dessus, le nom complet de *ClasseInterne* est :

  ```
  com.cgi.udev.ClasseEnglobante.ClasseInterne
  ```

- La classe englobante et la classe interne partage le même espace privé. Cela signifie que les attributs et les méthodes privés déclarés dans la classe englobante sont accessibles à la classe interne. Réciproquement, la classe englobante peut avoir accés aux éléments privés de la classe interne.
- Une instance de la classe interne n'a accès directement qu'aux attributs et aux méthodes de la classe englobante qui sont déclarés **static**.

Une classe interne **static** est souvent utilisée pour éviter de séparer dans des fichiers différents de petites classes utilitaires et ainsi de faciliter la lecture du code. Dans l'exemple ci-dessous, plutôt que de créer un fichier spécifique pour l'implémentation d'un comparateur, on ajoute son implémentation comme une classe interne.

```
package com.cgi.udev;
import java.util.Comparator;

public class Individu {

  public static class Comparateur implements Comparator<Individu> {
    @Override
    public int compare(Individu i1, Individu i2) {
      if (i1 == null) {
        return -1;
      }
      if (i2 == null) {
        return 1;
      }
      int cmp = i1.nom.compareTo(i2.nom);
      if (cmp == 0) {
        cmp = i1.prenom.compareTo(i2.prenom);
      }
      return cmp;
    }
  }

  private final String prenom;
  private final String nom;
```

```
  public Individu(String prenom, String nom) {
    this.prenom = prenom;
    this.nom = nom;
  }

  @Override
  public String toString() {
    return this.prenom + " " + this.nom;
  }

}
```

```
Individu[] individus = {
    new Individu("John", "Eod"),
    new Individu("Annabel", "Doe"),
    new Individu("John", "Doe")
};

Arrays.sort(individus, new Individu.Comparateur());

System.out.println(Arrays.toString(individus));
```

Dans l'exemple ci-dessus, la classe *Individu* fournit publiquement une implémentation d'un Comparator qui permet de comparer deux instances en fonction de leur nom et de leur prénom. Notez que l'implémentation de la méthode *compare* peut accéder aux attributs privés *nom* et *prenom* des paramètres *i1* et *i2* car ils sont de type *Individu*.

Les classes internes

Une classe interne qui n'est pas déclarée avec le mot-clé **static** est liée au contexte d'exécution d'une instance de la classe englobante.

Comme pour les classes internes **static**, le nom complet de classe interne inclus celui de la classe englobante et les deux classes partagent le même espace privé. Mais surtout, une classe interne maintient une référence implicite sur un objet de la classe englobante. Cela signifie que :

- une instance d'une classe interne ne peut être créée que par un objet de classe englobante : c'est-à-dire dans le corps d'une méthode ou dans le corps d'un constructeur

de la classe englobante.
- une instance d'une classe interne a accès directement aux attributs de l'instance dans le contexte de laquelle elle a été créée.

Une classe interne est utilisée pour créer un objet qui a couplage très fort avec un objet du type de la classe englobante. On utilise fréquemment le mécanisme de classe interne lorsque l'on veut réaliser une interface graphique en Java avec l'API Swing.

```
package com.cgi.udev;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;

import javax.swing.AbstractAction;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;

public class BoiteDeDialogue extends JDialog {

  private class IncrementerAction extends AbstractAction {
    public IncrementerAction() {
      super("Incrémenter");
    }

    @Override
    public void actionPerformed(ActionEvent e) {
      incrementer();
    }
  }

  private class DecrementerAction extends AbstractAction {
    public DecrementerAction() {
      super("Décrémenter");
    }

    @Override
    public void actionPerformed(ActionEvent e) {
      decrementer();
    }
```

```
  }

  private JLabel label;
  private int valeur;

  @Override
  protected void dialogInit() {
    super.dialogInit();
    this.setLayout(new FlowLayout());
    this.label = new JLabel(Integer.toString(this.valeur));
    this.add(this.label);
    this.add(new JButton(new IncrementerAction()));
    this.add(new JButton(new DecrementerAction()));
    this.pack();
  }

  private void incrementer() {
    label.setText(Integer.toString(++this.valeur));
  }

  private void decrementer() {
    label.setText(Integer.toString(--this.valeur));
  }

  public static void main(String[] args) {
    BoiteDeDialogue boiteDeDialogue = new BoiteDeDialogue();
    boiteDeDialogue.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    boiteDeDialogue.setVisible(true);
  }

}
```

L'exemple ci-dessus est un programme complet qui crée une boite de dialogue contenant deux boutons qui permettent respectivement d'incrémenter et de décrémenter un nombre qui est affiché. La classe JButton qui représente un bouton attend comme paramètre de construction une instance implémentant l'interface Action. Cette instance définit le libellé du bouton et l'action à réaliser lorsque l'utilisateur clique sur le bouton. Les boutons sont créés aux lignes 44 et 45. Les classes d'action utilisées pour chaque bouton sont définies aux lignes 13 et 24. Ces classes sont des classes internes. Dans leur méthode actionPerformed, elles appellent soit la méthode *incrementer* soit la méthode *decrementer*. Ces deux méthodes sont définies par la classe

englobante BoiteDeDialogue. Donc les instances de ces classes d'action appellent ces méthodes sur l'instance de l'objet englobant qui les a créées. Ainsi, les classes internes possèdent une référence sur l'objet BoiteDeDialogue qui les a créées.

> Notez dans l'exemple ci-dessus que les méthodes *BoiteDeDialogue.incrementer* et *BoiteDeDialogue.decrementer* sont privées. Comme une classe interne partage la même portée que sa classe englobante alors les classes internes *IncrementerAction* et *DecrementerAction* peuvent appeler ces méthodes.

Les classes anonymes

Une classe anonyme est une classe qui n'a pas de nom. Elle est déclarée au moment de l'instanciation d'un objet. Comme une classe anonyme n'a pas de nom, il n'est pas possible de déclarer une variable qui serait un type de cette classe. Une classe anonyme est donc utilisée pour créer à la volée une classe qui spécialise une autre classe ou qui implémente une interface. Pour déclarer une classe anonyme, on déclare le bloc de la classe au moment de l'instantiation avec **new**.

Imaginons que nous souhaitions créer une interface pour représenter un système de log :

```
package com.cgi.udev.logger;

public interface Logger {

  void log(String message);

}
```

On peut fournir une classe *GenerateurLogger* qui crée des instances implémentant l'interface *Logger*.

```
package com.cgi.udev.logger;

import java.time.LocalDateTime;

public class GenerateurLogger {

  private String application;

  /**
```

```
     * @param application Le nom de l'application
     */
    public GenerateurLogger(String application) {
      this.application = application;
    }

    public Logger creerConsoleLogger() {
      return new Logger() {
        @Override
        public void log(String message) {
          // Pour le format du message utilisé dans printf
          // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
          System.out.println(String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                                           LocalDateTime.now(), application, message));
        }
      };
    }

}
```

L'implémentation de la méthode *creerConsoleLogger* crée une instance implémentant l'interface *Logger* à partir d'une classe anonyme. L'implémentation de la méthode *log* affiche sur la sortie standard une chaîne de caractères formatée contenant la date et l'heure, le nom de l'application et le message passé en paramètre. Le nom de l'application correspond à l'attribut *application* de la classe *GenerateurLogger*. Comme pour les classes internes, les classes anonymes ont accès aux attributs et aux méthodes de l'objet englobant.

Il est possible de récupérer un objet implémetant *Logger* :

```
GenerateurLogger generateur = new GenerateurLogger("mon_appli");
Logger logger = generateur.creerConsoleLogger();
logger.log("un message de log");
```

Le code précédent affichera sur la sortie standard :

```
2017-nov.-jeu. 15:58 mon_appli - un message de log
```

Nous pouvons enrichir notre implémentation. Par exemple, la classe *GenerateurLogger* peut créer un logger qui ne fait rien ou encore un logger qui écrit les messages dans un fichier.

```
package com.cgi.udev.logger;
```

```
import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;

public class GenerateurLogger {

 private String application;

 /**
  * @param application Le nom de l'application
  */
 public GenerateurLogger(String application) {
  this.application = application;
 }

 public Logger creerConsoleLogger() {
  return new Logger() {
   @Override
   public void log(String message) {
    // Pour le format du message utilisé dans printf
    // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
    System.out.println(String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                                     LocalDateTime.now(), application, message));
   }
  };
 }

 public Logger creerNoopLogger() {
  return new Logger() {
   @Override
   public void log(String message) {
   }
  };
 }

 public Logger creerFileLogger(Path path) {
```

```
    return new Logger() {
      @Override
      public void log(String message) {
        // Pour le format du message utilisé dans printf
        // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
        String logMessage = String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                                          LocalDateTime.now(), application, message);
        try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE, StandardOpenOption.APPEND))
{
          w.append(logMessage).append('\n');
        } catch (IOException e) {
          System.err.println(logMessage);
        }
      }
    };
  }

}
```

La classe ci-dessus définit maintenant trois classes anonymes qui implémentent toutes l'interface *Logger*. Notez à la ligne 50, que la classe anonyme qui écrit le message de log dans un fichier, ouvre le fichier à partir d'un paramètre *path* passé à la méthode *creerFileLogger*. Cela signifie qu'une classe anonyme a accès au paramètre de la méthode qui la déclare.

> Une classe anonyme peut utiliser les paramètres et les variables de la méthode qui la déclare uniquement à condition qu'ils ne soient modifiés ni par la méthode ni par la classe anonyme. Avant Java 8, le compilateur exigeait que ces paramètres et ces variables soient déclarés avec le mot-clé **final**. Même s'il n'est plus nécessaire de déclarer explicitement le statut **final**, le compilateur générera tout de même une erreur si on tente de modifier un paramètre ou une variable déclaré dans la méthode et utilisé par une classe anonyme.

```
// on déclare le paramètre final pour signaler explicitement qu'il n'est
// pas possible de modifier la référence de ce paramètre puisqu'il est
// utilisé par la classe anonyme.
public Logger creerFileLogger(final Path path) {
  return new Logger() {
    @Override
    public void log(String message) {
      // Pour le format du message utilisé dans printf
      // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
```

```
      String logMessage = String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                                        LocalDateTime.now(), application, message);
      try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE, StandardOpenOption.APPEND)) {
        w.append(logMessage).append('\n');
      } catch (IOException e) {
        System.err.println(logMessage);
      }
    }
  };
}
```

Accès aux éléments de l’objet englobant

Si nous reprenons notre code de la classe *GenerateurLogger*, nous nous rendons compte que le formatage du message a été dupliqué pour le logger qui écrit sur la sortie standard et pour celui qui écrit dans un fichier. Afin de mutualiser le code, nous pouvons créer une méthode *genererLogMessage* dans la classe englobante qui pourra être appelée par chaque classe anonyme.

```
package com.cgi.udev.logger;

import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;

public class GenerateurLogger {

  private String application;

  /**
   * @param application Le nom de l'application
   */
```

```
  public GenerateurLogger(String application) {
    this.application = application;
  }

  public Logger creerConsoleLogger() {
    return new Logger() {
      @Override
      public void log(String message) {
        // Pour le format du message utilisé dans printf
        // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax

        System.out.println(genererLogMessage(message));
      }
    };
  }

  public Logger creerFileLogger(Path path) {
    return new Logger() {
      @Override
      public void log(String message) {
        try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE, StandardOpenOption.APPEND))
{
          w.append(genererLogMessage(message)).append('\n');
        } catch (IOException e) {
          System.err.println(genererLogMessage(message));
        }
      }
    };
  }

  public Logger creerNoopLogger() {
    return new Logger() {
      @Override
      public void log(String message) {
      }
    };
  }

  private String genererLogMessage(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
```

```
                LocalDateTime.now(), application, message);
  }


}
```

Mais nous voulons appeler cette nouvelle méthode *log*. Ce nom rentrera en collision avec le nom de le méthode *log* de l'interface *Logger*. Il existe une syntaxe particulière qui permet de référencer explicitement le contexte de la classe englobante en utilisant :

```
NomDeLaClasse.this
```

Ainsi nous pouvons renommer notre méthode *genererLogMessage* en *log* et nous pouvons l'invoquer explicitement dans les méthodes des classes anonymes avec la syntaxe :

```
GenerateurLogger.this.log(message);
```

> Cette syntaxe permet d'accéder aux attributs et aux méthodes de l'instance de la classe englobante.

```
package com.cgi.udev.logger;

import java.io.IOException;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.time.LocalDateTime;

public class GenerateurLogger {

  private String application;

  /**
   * @param application Le nom de l'application
   */
  public GenerateurLogger(String application) {
    this.application = application;
  }
```

```
  public Logger creerConsoleLogger() {
    return new Logger() {
      @Override
      public void log(String message) {
        // Pour le format du message utilisé dans printf
        // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
        System.out.println(GenerateurLogger.this.log(message));
      }
    };
  }

  public Logger creerFileLogger(Path path) {
    return new Logger() {
      @Override
      public void log(String message) {
        // Pour le format du message utilisé dans printf
        // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
        try(Writer w = Files.newBufferedWriter(path, StandardOpenOption.CREATE, StandardOpenOption.APPEND))
{
          w.append(GenerateurLogger.this.log(message)).append('\n');
        } catch (IOException e) {
          System.err.println(GenerateurLogger.this.log(message));
        }
      }
    };
  }

  public Logger creerNoopLogger() {
    return new Logger() {
      @Override
      public void log(String message) {
      }
    };
  }

  private String log(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                         LocalDateTime.now(), application, message);
  }
```

```
}
```

> Il n’est pas possible de déclarer un constructeur dans une classe anonyme. En effet, un constructeur porte le même nom que sa classe et justement, par définition, les classes anonymes n’ont pas de nom. Le compilateur générera néanmoins un constructeur par défaut.

Cela entraîne une limitation : il n’est pas possible de déclarer une classe anonyme qui étendrait une classe ne possédant pas de constructeur sans paramètre.

Classe interne à une méthode

Il est possible de déclarer une classe dans une méthode. Dans ce cas, il n’est pas possible de préciser la portée de la classe. La classe a automatiquement une portée très particulière puisqu’elle n’est visible que depuis la méthode dans laquelle elle est déclarée. Une classe déclarée dans une méthode peut fonctionner de la même manière qu’une classe anonyme : elle peut accéder aux paramètres et aux variables de la méthode qui la déclare (à condition qu’ils ne soient modifiés ni par la méthode ni par la classe).

```
package com.cgi.udev.logger;

import java.time.LocalDateTime;

public class GenerateurLogger {

  private String application;

  /**
   * @param application Le nom de l'application
   */
  public GenerateurLogger(String application) {
    this.application = application;
  }

  public Logger creerConsoleLogger() {
    class ConsoleLogger implements Logger {
      @Override
      public void log(String message) {
```

```
        // Pour le format du message utilisé dans printf
        // Cf. https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax
        System.out.println(GenerateurLogger.this.log(message));
      }
    }
    return new ConsoleLogger();
  }

  private String log(String message) {
    return String.format("%1$tY-%1$tb-%1$ta %1$tH:%1$tM %2$s - %3$s",
                         LocalDateTime.now(), application, message);
  }

}
```

Dans l'exemple ci-dessus, la méthode *creerConsoleLogger* déclare la classe interne *ConsoleLogger*.

> Contrairement aux classes anonymes, une classe interne à une méthode peut déclarer des constructeurs.

Interface et énumération

Il est possible de déclarer des interfaces et des énumérations dans une classe. Il est même possible de déclarer des interfaces et des énumérations dans une interface. Dans ce cas, les interfaces et les énumérations sont traitées implicitement comme **static**. On peut ou non préciser le mot-clé.

```
package com.cgi.udev;

public class ClasseEnglobante {

  public interface InterfaceInterne {

  }

  public enum EnumerationInterne{VALEUR1, VALEUR2}

}
```

Plusieurs classes dans un même fichier

Même s’il ne s’agit pas de classes internes, il est possible de déclarer plusieurs classes dans un même fichier en Java. Mais les classes supplémentaires sont forcément de portée package.

En pratique cette possibilité n’est jamais utilisée par les développeurs qui préfèrent utiliser des classes internes **static** ou un fichier propre à chaque classe.

Revision #1
Created 10 February 2025 22:27:41 by Nicolas
Updated 10 February 2025 22:29:25 by Nicolas