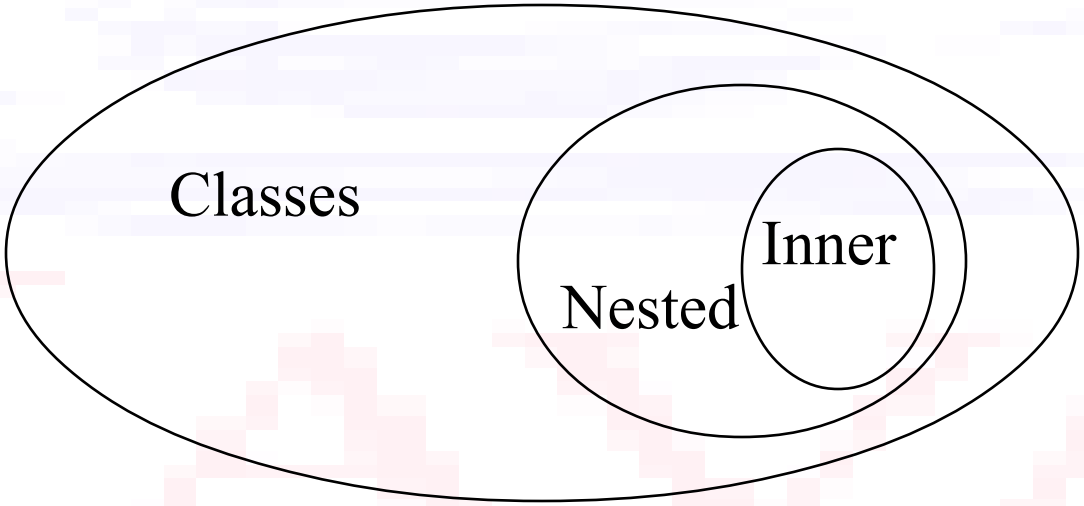
Classes Internes

Classes Internes

- Une classe Java peut contenir
  - Variables
    - d'instance
    - **`static`**
  - Méthodes
    - d'instance
    - **`static`**
  - ...mais aussi des *classes*
    - d'instance
    - **`static`**

Taxonomie

- *Nested classes*
  - Classes imbriquées
- *Inner classes*
  - Classes internes



Classes Internes

- Souvent liées aux...
  - évènements
  - JavaBeans
- ...mais peuvent être employées n'importe où
- Améliorent la cohérence de la syntaxe

Taxonomie

- Static nested classes
- Inner classes
  - Member classes
  - Local classes
  - Anonymous classes

Taxonomie

- Classe *top-level*
  - Contenue dans un package

```
package foo;
public class Toto {
   ...
}

class Titi {
   ...
}
```

- *Nested* classe
  - Contenue dans une classe

```
package foo;
public class Toto {
   ...
   class Titi {
      ...
   }
   ...
}
```

Taxonomie

- Nested - notion purement syntaxique
  - Une classe est membre d'une autre classe

```
public class Toto {
   ...
   class Titi {
      ...
   }
   ...
}
```

Nested

- Au même titre qu'une...
  - variable d'instance
  - méthode
- Mêmes prérogatives
  - Niveaux d'accès
  - Visibilité
    - accède même aux membres `private` de la classe englobante

Nested

- Peuvent être `static`
  - On parle de *static nested* classes
- Si non-`static`
  - On parle de *inner* classes

Nested

- Static nested classes
  - Liée à la classe `Toto`

```
public class Toto {
   ...
   static class Titi {
      ...
   }
   ...
}
```

```
Toto.Titi tt
  = new Toto.Titi();
```

- Inner classes
  - Liée à un objet de type `Toto`

```
public class Toto {
   ...
   class Titi {
      ...
   }
   ...
}
```

```
Toto.Titi tt
  = new Toto().new Titi();
```

Static Nested

- Aucun accès aux membres non-static de la classe englobante
- Référencée par rapport à la classe englobante
- Façon de lier des classes...liées
- Interfaces nested sont implicitement static

```
public class Foo extends Toto.Titi {…}

public static class Ellipse2D.Float extends Ellipse2D
public static class Ellipse2D.Double extends Ellipse2D
```

Inner classe

- Classe non static nested
- Liée à un objet instance de la classe englobante
- Une inner classe peut être
  - Membre
    - correspond aux méthodes et variables d'instances
  - Local
    - correspond aux variables locales
  - Anonyme

Les Classes Membres

- Similaires aux variables et méthodes d'instance
- Accès à tous les membres de la classe englobante
  - Même aux membres `private`
- La classe englobante a accès à tous les membres d'une classe membre
  - Même aux membres `private`
- Chaque instance est associée avec une instance de la classe englobante
  - Comme pour les méthodes et les variables

Utilisation

- Partout où la classe membre est liée à la classe englobante
  - Pour des classes auxiliaires
  - Pour des adaptateurs d'évènements

Exemple de Classe Auxiliaire

- Quel sera le résultat ?

```
public class TotoEtHelper {
  private int count = 0;
  public TotoEtHelper() { // constructeur
    this.new Helper();
    System.out.print(count + " fois...");
    this.new Helper();
    System.out.println(count + " fois");
  }

  private class Helper {
    private Helper() { count++; }
  }
}
```

Exemple de Classe Auxiliaire

```
prompt: java TotoEtHelper
1 fois...2 fois
prompt:
prompt: ls Toto*class
TotoEtHelper$Helper.class  TotoEtHelper.class
prompt:
```

Les Classes Locales

- Similaires aux méthodes et variables locales
- Déclarées localement dans un bloc de code
  - Dans une méthode
  - Dans un bloc initialisateur
- Utilisation similaire aux classes membres

Les Classes Anonymes

- Classe locale sans nom
- Définition et instanciation combinées

```
public class Toto {
   ...
   public Titi uneMethode() {
      ...
      return new Titi() { // création de classe anonyme
        // définition de Titi
      }
   }
   ...
}
```

Syntaxe

- Classe

```
new nom-de-classe([liste d'args]) {
  défn-de-classe
}
```

```
toto = faisToto(new Titi(arg1, arg2) {
  //défn de la classe anonyme
});
```

  - la classe anonyme hérite de **`Titi`**
  - **`arg1`**, **`arg2`** passées au constructeur de **`Titi`**
  - classe anon. n'a pas de nom --> pas de constructeur
  - pas d'interface - aucun **`implements`** possible

Syntaxe

- Interface

```
new nom-d'interface() {
  défn-de-l'interface
}
```

  - implante l'interface nommée
  - hérite de **`Object`** - aucun **`extends`** possible

Utilisation

- Une classe anonyme peut convenir quand :
  - sa définition est brève
  - une seule instanciation
  - un constructeur n'est pas nécessaire
  - utilisée tout de suite après sa définition
  - un nom ne contribue rien à la compréhension du code
- Ne pas en abuser !

Utilisation

- Le plus souvent pour réagir aux évènements

```
public FrameTest extends JFrame {
  public static void main(java.lang.String[] args) {
    final FrameTest ft = new FrameTest("Frame Test");
    ft.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        ft.dispose();
        System.exit(0);
      }
    });
    ft.setSize(250, 150);
    ft.setVisible(true);
  }
}
```