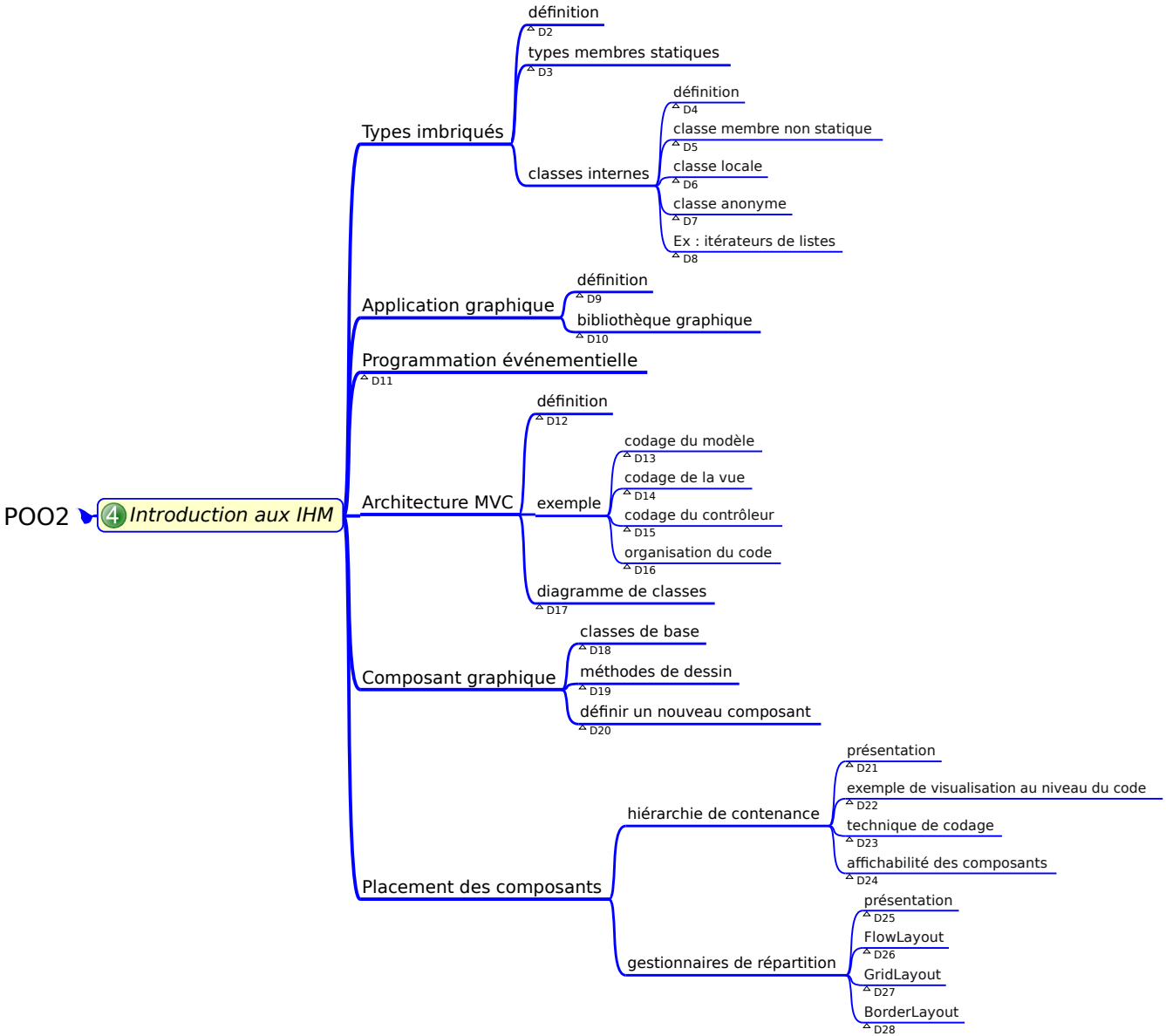
Applications graphiques Java



Applications graphiques Java

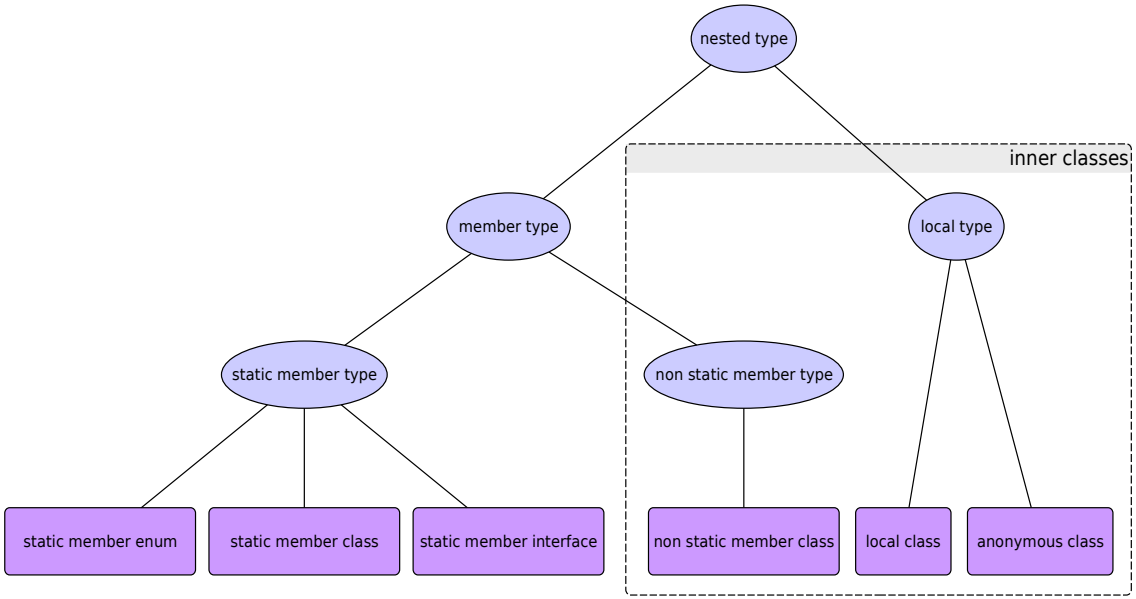
Type imbriqué :
Type déclaré à l'intérieur d'un autre type.

Type englobant :
Type contenant la déclaration d'un autre type.

type englobant

types imbriqués

```
class A {
    class B { /* def */ }
    void m() {
        class C { /* def */ }
        C x = new C();
        ...
    }
}
```



Applications graphiques Java

Type membre statique : Type imbriqué, déclaré **`static`** au même niveau que les membres de son type englobant.

```
class A {
    static int i;
    static void m() { ... }
    int j;
    void p() { ... }
    acces static categ B {
        // accès à i et m()
        // mais pas à j ni à p()
    }
}
```

acces = **public** | **protected** | *<rien>* | **private**
categ = **enum** | **interface** | **class**

Rq : sont implicitement déclarés statiques
- tout type imbriqué dans une interface
- toute interface ou type énuméré imbriqué

```
class C {
    void test() {
        A.B x = new A.B()
    }
}
```

type englobant = espace de nommage

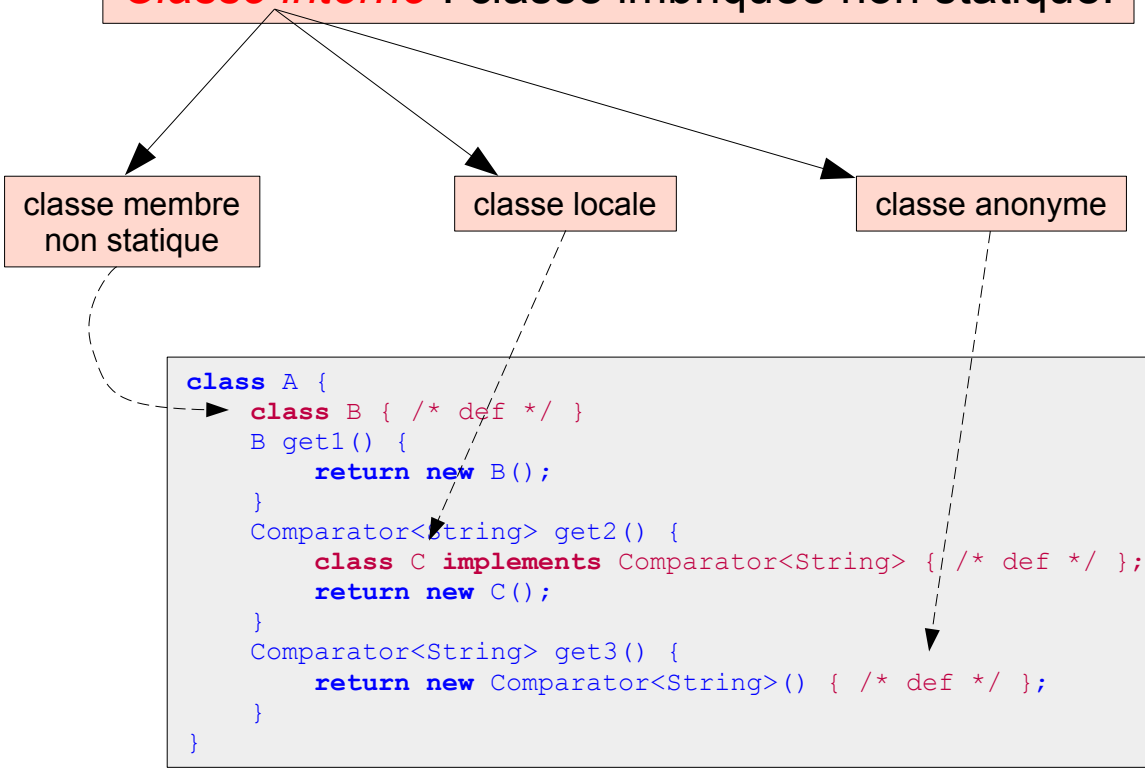
accès à `B` en dehors de `A` : `A.B`
(nom simple qualifié par le nom du type englobant)

```
public interface Map<K, V> {
    [public static] interface Entry<K, V> {
        ...
    }
    ...
}
```

```
public class HashMap<K, V> implements Map<K, V> {
    private Set<Map.Entry<K, V>> entrySet;
    ...
    static class Entry<K, V> implements Map.Entry<K, V> {
        ...
    }
}
```

Applications graphiques Java

Classe interne : classe imbriquée non statique.



Nom compilé / Nom dans le code source

```
a.get1().getClass().getName()  == A$B
a.get2().getClass().getName()  == A$1C
a.get3().getClass().getName()  == A$2
```

`B` dans le texte de `A`
`A.B` ailleurs dans le code source

`C` dans le corps de `get2`
inaccessible ailleurs dans le code source

inaccessible dans le code source

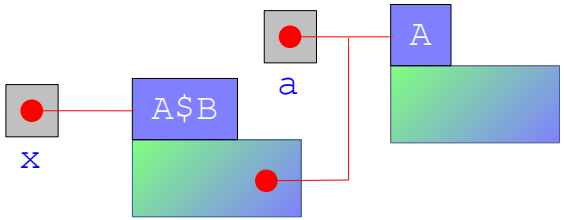
les noms `A$...` sont synthétisés par le compilateur et inutilisables dans le code source

Classe membre non statique : classe interne déclarée au même niveau que les membres de sa classe englobante.

```
class A {
    acces class B { /* def */ }
}
```

acces = **public** | **protected** | *<rien>* | **private**

La création d'une instance de classe interne **nécessite** la présence d'une instance de la classe directement englobante.




```
class C {
    void test(A a) {
        A.B x = a.new B();
    }
}
```

Accès aux différentes caractéristiques

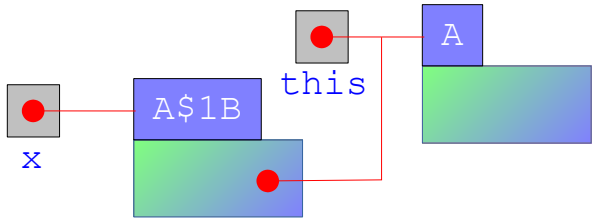
```
class A {
    int i;
    static int j;
    class B {
        int k;
        void m() {
            ... [A.this.]i ...
            ... [A.]j ...
            ... [[B.]this.]k ...
        }
    }
}
```

Classe locale : classe interne déclarée dans un bloc de code de sa classe englobante.

```
class A {
    void m() {
        class B { /* def */ };
        ...
    }
}
```

La création d'une instance de classe locale **utilise implicitement** `this` comme instance directement englobante*.

* : sauf si cette création se fait dans un contexte statique




```
class A {
    void m() {
        class B { /* def */ };
        B x = new B();
        ...
    }
}
```

Accès aux différentes caractéristiques

```
class A {
    int i;
    static int j;
    void m(final int a) {
        final int b;
        class B {
            int k;
            void p() {
                ... a ... b ...
                ... [A.this.]i ...
                ... [A.]j ...
                ... [[B.]this.]k ...
            }
        }
    }
}
```

```
class A {
    int i;
    static int j;
    static void m(final int a) {
        final int b;
        class B {
            int k;
            void p() {
                ... a ... b ...
                ... [A.this.]i ...
                ... [A.]j ...
                ... [[B.]this.]k ...
            }
        }
    }
}
```

Applications graphiques Java

Classe anonyme : classe locale sans nom, dotée d'un unique constructeur implicite, instanciée en même temps qu'elle est définie.

Forme syntaxique spéciale :

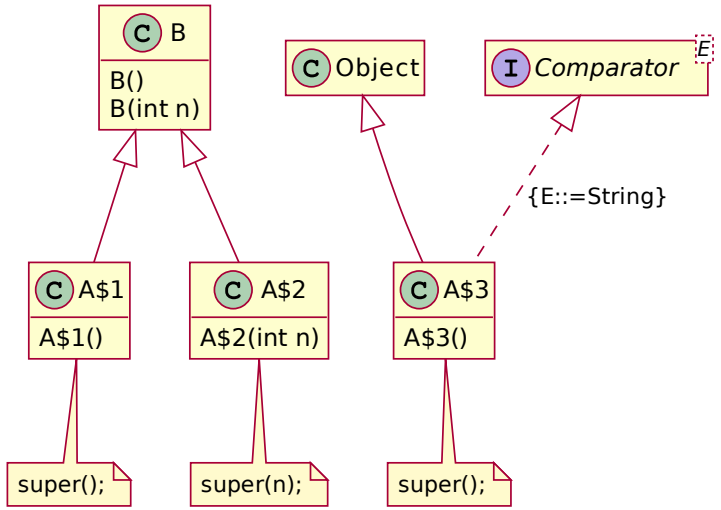
```
Type var = new Type([args]) {
    /* définition sans constructeur */
};
```

produit une instance d'un sous-type (anonyme) direct de *Type*

CT(x) = B
RT(x) = A$1

```
class A {
    void m() {
        B x = new B() {
            // définition de A$1
        };
        B y = new B(5) {
            // définition de A$2
        };
        Comparator<String> z = new Comparator<String>() {
            // définition de A$3
        };
    }
}
```

détermine la signature et le corps du constructeur



Applications graphiques Java


```
public class AbstractList<E> implements List<E> {
    protected int modCount = 0;
    private class Itr implements Iterator<E> {
        ...
        int expectedModCount = modCount;
        ...
        public E next() {
            checkForComodification();
            ...
        }
        ...
        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new CME();
        }
    }
    public Iterator<E> iterator() {
        return new Itr();
    }
    ...
}
```

compte le nombre de modifications de la liste

compteur de l'instance englobante courante

```
public class ArrayList<E> extends AbstractList<E> {
    public boolean add(E e) {
        modCount++;
        ...
    }
    public E remove(int i) {
        modCount++;
        ...
    }
    ...
}
```

méthodes de l'instance englobante courante

accès *via* `AbstractList.this` nécessaire pour lever l'ambiguïté

```
private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastRet = -1;
    int expectedModCount = modCount;
    public boolean hasNext() {
        return cursor != size();
    }
    public E next() {
        checkForComodification();
        try {
            E next = get(cursor);
            lastRet = cursor++;
            return next;
        } catch (IOOBE e) {
            checkForComodification();
            throw new NSEE();
        }
    }
    public void remove() {
        if (lastRet == -1) throw new ISE();
        checkForComodification();
        try {
            AbstractList.this.remove(lastRet);
            if (lastRet < cursor) cursor--;
            lastRet = -1;
            expectedModCount = modCount;
        } catch (IOOBE e) {
            throw new CME();
        }
    }
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new CME();
    }
}
```