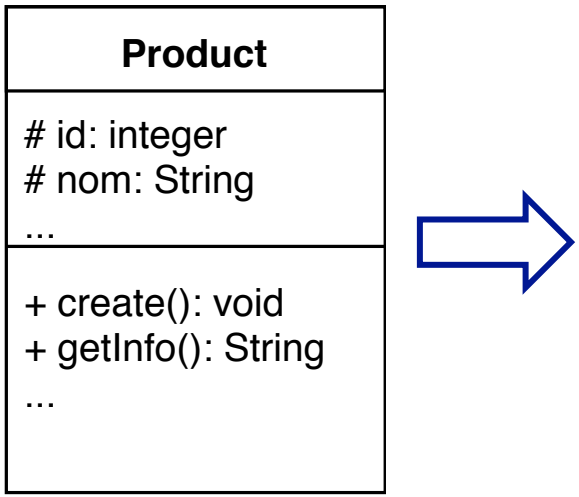
Plan

- Introduction au Java
  - Généralités
  - Syntaxe de base
- Concepts et modélisation orientée objets
  - Objet, classe et modélisation UML
  - Les principes fondamentaux: encapsulation, abstraction, héritage et polymorphisme
- Programmation Orientée Objets en Java
  - **Classes, objets et bonnes pratiques**
  - Héritage, interfaces, agrégation, composition et association
  - Packages
  - Généricité
  - Exceptions
  - Flux d'Entrée/Sortie et fichiers
  - Empaqueter et déployer son programme

Les classes en Java



Fichier Product.java

```
import java.util.*;

public class Product {

        protected int id;

        protected String nom;

        // …

        public void create() {
                // …
        }

        public String getInfo() {
                return id+"  : "+nom;
        }

        // …
}
```

les packages utilisés dans la classe

les attributs de la classe

les méthodes de la classe

Déclaration d'une classe

- [modifier] **class** *MyClass* [**extends** *MySuperClass*] [**implements** *YourInterfaces*]
  - *[modifier]* (optionnel): accessibilité de la classe dans les autres classes

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

  - *MyClass* : nom de la classe
  - ***[extends** MySuperClass]* (optionnel): nom de la classe mère (une seule)
  - ***[implements** YourInterfaces]* (optionnel): nom des interfaces implémentées

```
class MyClass
{
    // field, constructor, and
    // method declarations
}
```

```
public class MyClass extends MySuperClass implements YourInterface1, YourInterface2  {
    // field, constructor, and
    // method declarations
}
```

Déclaration d'un attribut dans une classe

- [modifier] [**static**] [**final**] *myType myVariable* [**=** *myDefaultValue*] **;**
  - *[modifier]* (optionnel): accessibilité de l'attribut dans les autres classes

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

  - ***[static]*** (optionnel): déclare une variable de classe
    - un attribut associé à la classe et non aux objets
  - ***[final]*** (optionnel): déclare une constante
  - *myType* : type de l'attribut
    - un type primitif ou une classe (p.ex. String)
    - si le type est une classe d'un autre package, faire un **import**
      n.b. String dans package java.lang mais importé par défaut

```
public class Product {

        protected int id = -1 ;

        protected String nom;


        public int getId() {
                return id;
        }

        public void setId( int newId ) {
                id = newId;
        }

        public String getNom() {
                return nom;
        }

        public void setNom( String newNom ) {
                nom = newNom ;
        }

}
```

- Accès aux attributs privés ou protégés via des méthodes dédiées (*setter/getter*)

Déclaration d'une méthode dans une classe

- [modifier] [**static**] *returnsType methodName***(** *paremeterList* **) { ⋯ }**
  - *[modifier]* (optionnel):

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

```
public class Product {
    // ⋯
    public boolean equals( Product prod ) {
        // ⋯
    }


    // ⋯
}
```

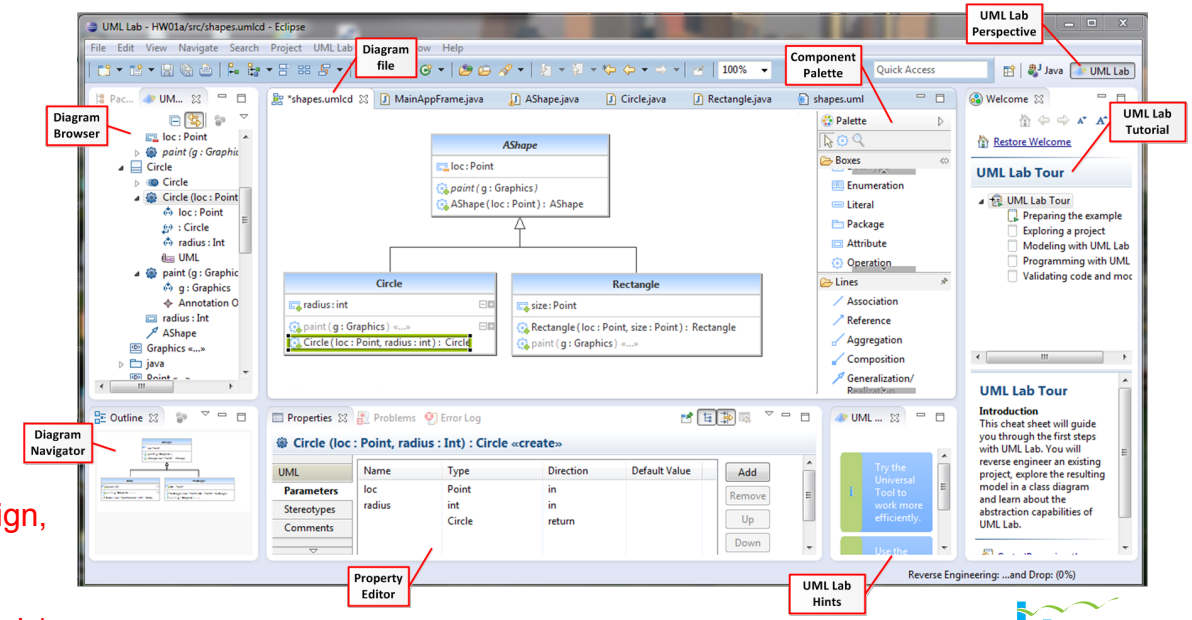
  - ***[static]*** (optionnel): déclaration d'une méthode de classe
  - *returnsType* : type de la valeur retournée, ou **void** si aucune valeur retournée
  - *methodName(paremeterList) { ⋯ }* **:** id. au langage C

Rq: **surcharge** de la méthode *equals*

Exercice: génération de code à partir du diagramme de classes

- Générer le code Java associé au modèle du logiciel de gestion de prêts dans des bibliothèques (UML Lab)
  1. Ouvrir le fichier contenant le diagramme de classes
     - fichier .umlcd
  2. Générer le code Java
     - *clic droit > Generate code*

     OU
     - menu *UML Lab > Code Generation > Generate code*
  3. Résoudre les erreurs de compilation, s'il y en a



Pour plus de détails cf.
Advanced Object-Oriented Programming and Design, Dr Stephen Wong, Rice University.

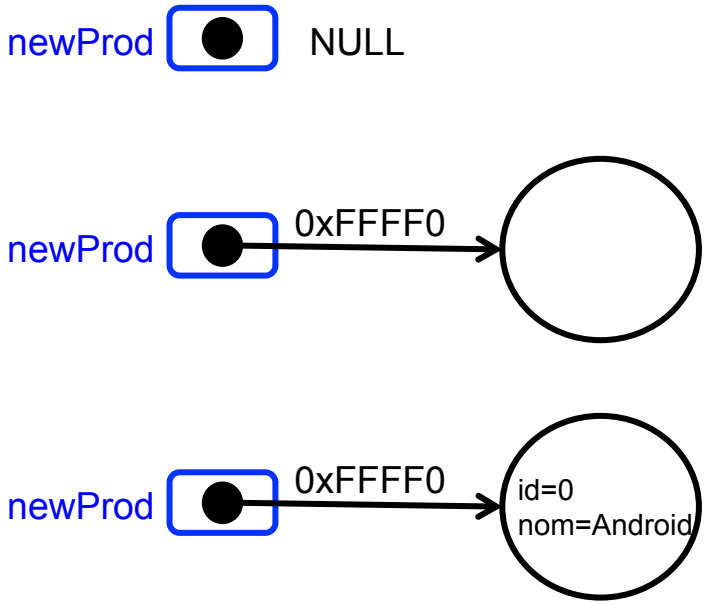
https://www.clear.rice.edu/comp310/Eclipse/UMLLab/

Créer un objet en Java

- Un programme Java créé beaucoup d'objets qui interagissent entre eux en invoquant des méthodes

```
ArrayList products;
// …
products = new ArrayList();


Product newProd = new Product(0, "Android") ;
```

- Une **création** en 3 étapes
  - La **déclaration**
    - association d'un nom de variable à un type (une classe)
  - L'**instanciation**
    - allocation de la mémoire associée à l'objet via l'opérateur **new**
  - L'**initialisation**
    - appel du constructeur (suite au **new**)
      méthode de la classe chargée d'initialiser les objets au moment de leur création



Intégrer un constructeur à la classe

- Constructeur = **méthode** de la classe chargée d'initialiser les objets au moment de leur création
  - Nom obligatoirement identique au nom de sa classe
  - Aucun type de retour

```
public class Product {

    protected int id;
    protected String nom;
    // …
    public Product() {
                id = -1;
                nom = "default";
                // …
    }

    public Product( int newId, String newNom ) {
                id = newId;
                nom = newNom;
                // …
    }
    // …
}
```

- Invocation:

```
Product p = new Product() ;
Product p2 = new Product( 2, "tetris");
```

  - Lors de la création d'un nouvel objet avec l'opérateur **new**

- Attention
  - Si pas de constructeurs définis -> utilisation par le compilateur du constructeur par défaut de la classe mère -> erreur s'il n'y en a pas
    - si pas de classe mère -> utilisation de celui de la class Object

Détruire un objet en Java

- Et la **destruction** (i.e. la libération de la mémoire associée à l'objet) ?
- Automatiquement faite par le "ramasse-miettes" de Java (***garbage collector***)
  - Tâche en arrière-plan qui intervient lorsque le système a besoin de mémoire ou périodiquement avec une priorité faible
  - Libère la place occupée par les **instances non référencées** et compacte la mémoire occupée
  - Libération de la mémoire plus ou moins "aléatoire" (libération asynchrone)

```
{
  Product myProd = new Product(2, "tetris") ;
  …
}
```

Fin d'une variable locale

OU

```
Product myProd = new Product(2, "tetris") ;
…
myProd = null ;
```

Aucune variable ne référence l'objet

- Rq.: possibilité de (re)définir une méthode appelée automatiquement au moment de la destruction
  - Méthode **finalize** d'**Object**

```
protected void finalize( ) throws Throwable
```

  - Utile pour libérer des ressources (p.ex. fermeture fichier, connexion BD, connexion réseau, …)

Utiliser les objets

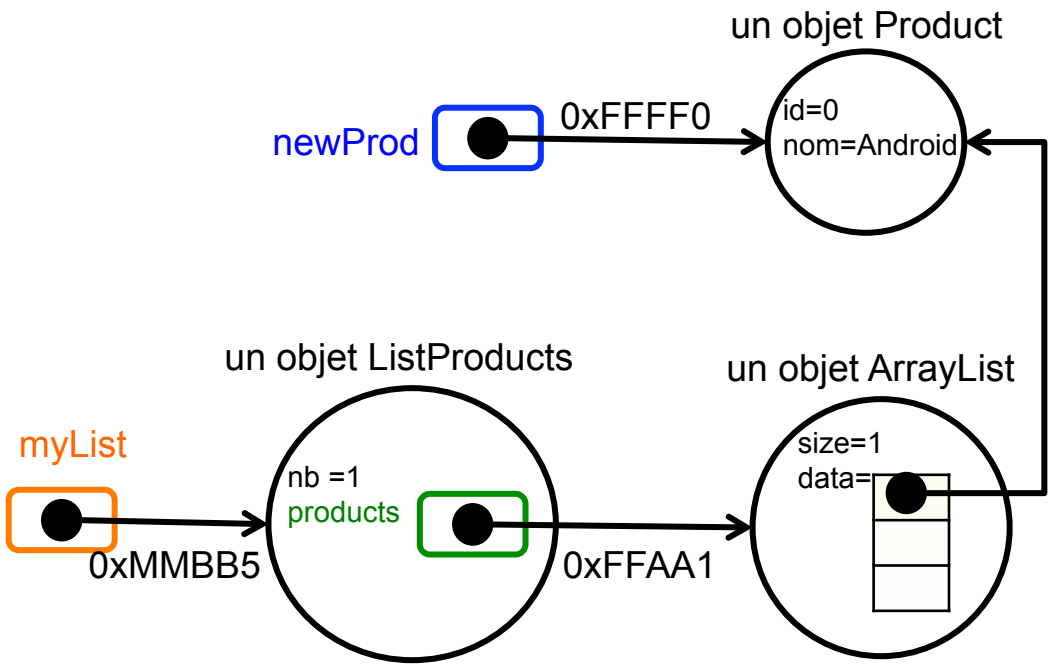
- Accéder aux attributs d'un objet: *objectReference.fieldName*
  - Uniquement possible en dehors de la classe si l'attribut est publique

```
public class Product {
    protected int id;
    // …
    public boolean equals( Product prod ) {
        return (id == prod.id);
    }
    // …
}
```

- Invoquer une méthode d'un objet:
  objectReference.methodName([*parameterList*])
  - Uniquement possible en dehors de la classe si la méthode est publique

```
import java.util.*;

public class ListProducts {

        private in nb = 0 ;
        private ArrayList products;
        // …
        public Product addProduct( int id, String nom ){
                Product newProd = new Product(id, nom) ;
                products.add( newProd );
                return newProd;
        }
        // …
}
```

Attention: objets en Java = pointeurs




```
import java.util.*;

public class ListProducts {

        private int nb = 0;
        private ArrayList products;
        // …

        public ListProducts() {
                products = new ArrayList();
        }

        public Product addProduct( int id, String nom ){
                Product newProd = new Product(id,nom) ;
                products.add( newProd );
                return newProd ;
        }
        // …
}
```

```
ListProducts myList = new ListProducts() ;
myList.addProduct( 0, "Android" ) ;
```

Attention: une modification de newProd après l'ajout dans myList engendre aussi une modification de myList

Passage des paramètres aux méthodes

- Si type du paramètre = type primitif
  - Passage par valeur (recopie)
  - ➢ Les valeurs des paramètres sont recopiées dans l'espace mémoire de la méthode (modifications non conservées en dehors de la méthode)
- Si type du paramètre = classe
  - Passage par référence
  - ➢ Si la méthode modifie l'objet passé en paramètre, les modifications seront conservées en dehors de la méthode

```
public class Banque {

    public void transfert( int ip, Compte c1p, Compte c2p) {
        ip ++ ;
        c2p.addSolde( c1p.getSolde() );
        c1p = new Compte() ;
    }


    public static void main( String[] args ){

        Compte c1 = new Compte("Jo", 10000);
        Compte c2 = new Compte("Al", 50000);

        int i = 0;
        transfert( i, c1, c2 ) ;

        System.out.println( i + " " + c1.getSolde() +" "+ c2.getSolde() );
    }
}
```

```
public class Compte {

    private String nomTitulaire = "";
    private double solde = 0;

    public Compte(String inNom double inSolde) {
        nomTitulaire = inNom ;
        solde = inSolde ;
    }

    public double getSolde() {
        return solde;
    }

    public void addSolde(double inSolde ){
        solde += inSolde ;
    }
}
```

➢ Affiche:
"0 10000 60000"

(et pas "1 0 60000")

Quelques exemples de classes et d'objets prédéfinis

- Les **tableaux**:
  - Des objets à part entière en Java, mais avec une syntaxe particulière pour leur déclaration et leur initialisation
  - Déclaration (une dimension): my*Type***[]** *myArray*
    - tableau à deux dimensions *myType***[][]** *myMatrix*
    - …
  - Création: **new** *myType***[***mySize***] ;**
    - taille obligatoire et non modifiable après
    - possibilité de créer et d'initialiser en même temps
  - Taille d'un tableau: attribut **length**
  - Java intègre des méthodes pour les manipulations courantes sur les tableaux
    - copier: **System.arrayCopy** ou **java.util.Arrays.copyOfRange,** etc
    - rechercher: **java.util.Arrays.binarySearch**
    - comparer: **java.util.Arrays.equals**
    - trier: **java.util.Arrays.parallelSort**
    - etc
  - Attention: ne pas utiliser les objets du tableau avant de les avoir créés

```
int[] anArray ;
anArray = new int[3] ;
anArray[0] = 2 ;
anArray[1] = 4 ;
anArray[2] = 2 ;

int[] anArrayCopy = {2, 4, 2} ;
```

```
String[][] names  = new String[3][2];
names[0][0] = "Aretha";
names[0][1] = "Franklin";
int nbLines = names.length ; // 3
int nbCol = names[0].length ; // 2

String[][] names2 = { {Bob, Marley}, {John,
Doe} };

String[][] namesCopy= new
                        String[names.length][];
for( int i=0; i < names.length; i++ )
   namesCopy[i] = Arrays.copyOf(names[i],
                              names[i].length) ;
```

```
Product[] arrayProd = new Product[100] ;
arrayProd[0] = new Product();
arrayProd[0].setNom("Firefox") ;
```

Quelques exemples de classes et d'objets prédéfinis

- Les **chaînes de caractères:**
  - Classe **String** (constantes, *immutable*)
    - affecter une nouvelle valeur: **= "**new text**";**
      (création d'un nouvel objet String)
    - copier une chaîne: **= new String(** *myStr* **);**
    - comparer deux chaînes: méthode **equals**
      attention: ne pas utiliser ==
    - taille de la chaîne: méthode **length()**
    - beaucoup de méthodes pour manipuler les chaînes: **charAt**, **concat,** opérateur **+**, **contains**, **format**, **indexOf**, **matches**, **replace**, **split**, **substring**, **trim**, **valueOf**, etc
    - attention aux performances pour certaines opérations et à la sécurité
      - si beaucoup de "modifications" (p.ex. concaténation ou **replace**) -> utiliser **StringBuilder** ou **StringBuffer**
      - ne pas stocker d'informations sensibles (p.ex. mots de passe) -> utiliser un **char[]**

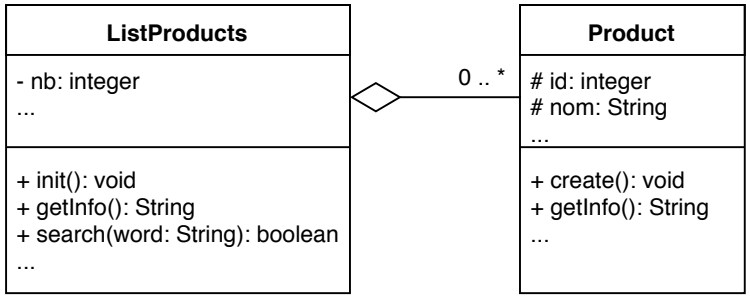
```
String helloStr1 = "hello";
String aliasStr1 = helloStr1;
String copyStr1 = new String(helloStr1);

char[] worldArray = {'w','o','r','l','d'};
String helloStr2 = new String( worldArray);

boolean isEqual = helloStr1.equals(worldArray);

helloStr1 = helloStr1 + " ";
helloStr1.concat( helloStr2 );
```

  - Classe **StringBuilder** (modifiables, *mutable*)
    - méthodes **append**, **delete**, **insert**, **replace**, **setCharAt**, **reverse** pour modifier
    - longueur et capacité réelle: nombre de caractères vs mémoire allouée
      - constructeur par défaut: 16 éléments vides
      - autres constructeurs: **StringBuilder(String s)**, **StringBuilder(int initCapacity)**, etc
      - capacité mise à jours automatiquement après des ajouts
    - une méthode **toString()** pour retourner un **String**

Exercice: Création de classes *Document* et *Client*

- A partir du modèle du logiciel de gestion des prêts de bibliothèques, compléter la classe *Document* avec les méthodes suivantes:
  - constructeur par défaut: *Document()*
  - constructeur permettant d'initialiser les attributs: *Document(code: String, titre: String, annee: int)*
  - méthode permettant de comparer si deux documents sont égaux: *equals(doc: Document): boolean*
  - méthode retournant une description du document: *toString():String*
  - méthode retournant si un document est empruntable: *estEmpruntable(): boolean*
    - dans un premier temps, ajouter un attribut booléen pour stocker si le document est disponible
- Créer une classe *DocumentTest* composée d'une méthode *main* et utiliser là pour tester les méthodes de votre classe *Document*
  - implémenter pour cela les méthodes de test: *testEquals():boolean*, *testToString():boolean*
- Faire de même pour la classe *Client* en remplaçant la dernière méthode par une méthode *peutEmprunter(): boolean* retournant si un client peut emprunter un document

Le mot clé **this**



- Utiliser **this** avec un attribut
  - Référence l'objet courant à l'intérieur d'une méthode
    - utile lorsque des paramètres ont le même nom que les attributs

- Utiliser **this** avec un constructeur
  - Appeler un autre constructeur de la même classe
    - limite la duplication du code
    - attention: invocation nécessairement en première ligne du constructeur

```
public class Product {

    int id;
    String nom;
    // …
    public Product( int id, String nom ) {
                this.id = id;
                this.nom = new String(nom);
                // …
   }
   // …
}
```

```
import java.util.*;

public class ListProducts {

        private int nb = 0 ;
        private ArrayList products;
        // …
        public ListProducts() {
                products = new ArrayList();
        }

        public ListProducts( Product prod ){
                this();
                products.add( prod );
        }
        // ...
}
```

Les variables et les méthodes de classe (**static**)

- Variable (resp. méthode) associée à une classe et non à une instance de cette classe
  - Commun à tous les objets instanciés
    - mais utilisable sans création d'objets
  - Accès: *MyClass.staticField* ou *MyClass.staticMethod*

```
public class Product {

    protected int id;
    protected String nom;
    protected static int nbProducts = 0 ;
    // …
    public Product( int id, String nom ) {
                this.id = id;
                this.nom = new String(nom);
                nbProducts++ ;
                // …
    }

    public static int getNbProducts() {
                return nbProducts ;
    }
    // …
}
```

```
int arr1[] = {0,1,2,6,7} ;
int arr2[] = {3,4,5} ;
System.arraycopy(arr2,0,arr1,3,2);
// résultat {0,1,2,3,4} ;

int nb = Product.getNbProducts();
```

Attention: les méthodes de classe ne peuvent pas

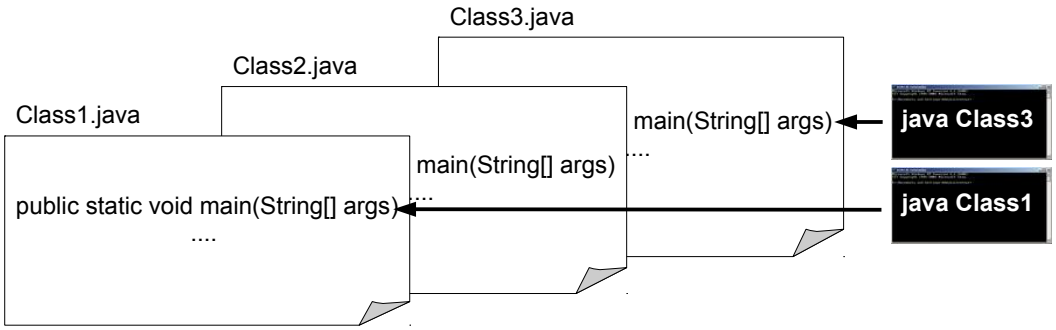
- Accéder directement aux méthodes et aux attributs des instances
  - nécessité de passer par un objet
- Utiliser **this**

La méthode main

- Point d'entrée pour l'exécution d'une application Java

- Méthode statique de la classe exécutée par la machine virtuelle
  - **public static void main(String[]** args**)**

- Plusieurs classes d'une même application peuvent contenir leur propre méthode **main**
  - La méthode main à exécuter est celle de la classe indiquée à la JVM



- Intérêt: possibilité de définir de manière indépendante des tests pour chacune des classes d'un logiciel

Les classes imbriquées (*nested classes*)

```
class OuterClass {
    …
    class NestedClass {
        …
    }
}
```

- Définition d'une classe dans une autre classe
  - **classe interne** (*inner class*, non statique)
    - accès à tous les autres membres de *OuterClass* (même si privés)
    - notion de *helper class*
  - **classe statique**
    - accès uniquement aux membres statiques

- Intérêt:
  - Regrouper des classes qui sont uniquement utilisées à un seul endroit
  - Augmenter l'encapsulation
  - Générer un code plus lisible, concis et maintenable
    - code plus proche de là où il est utilisé

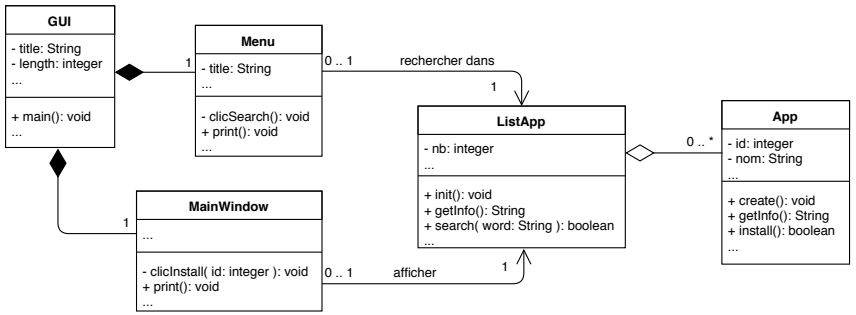
```
import java.util.*;
public class ListProducts {
        private int nb = 0 ;
        private ArrayList products;
        private class ListProductsIterator {
            private int nextIndex = 0 ;
            public boolean hasNext() {
                return ( nextIndex < nb ) ;
            }
            public String next() {
                nextIndex +=1;
                return products.get(nextIndex-1);
            }
        }
        // …
        public String getInfo(){
                String info = "";
                ListProductsIterator iterator =
                        this.new ListProductsIterator() ;
                while( iterator.hasNext() ){
                    info += iterator.next() + " ";
                }
                return info ;
        }
        // …
}
```

Les classes imbriquées (*nested classes*)

- Possibilité de définir une classe interne dans une **méthode**
  - Accès à tous les membres de *OuterClass* + les constantes locales (et variables non modifiées) de la méthode + paramètres de la méthode
  - Deux types:
    - **classe locale** (non statique avec un nom)
    - **classe anonyme**, i.e. classe locale sans nom
      - déclarée et instanciée en même temps
      - très utilisée pour les interfaces graphiques

nom d'une interface du package awt.event

```
import java.awt.event.*;
…
public class MainWindow extends JFrame{
    // …
    public MainWindow() {
        int id = 0;
        // …
        installButton.addActionListener(
                new ActionListener() {
                    public void actionPerformed( ActionEvent evt) {
                        clicInstall( id );
                    }
                }
        );
        // …
    }

    private void clicInstall( int id ) {
        // …
    }
}
```



Les expressions Lambda

- Limite des classes anonymes:
  - Syntaxe pas forcément simple et clair lorsqu'il s'agit d'implémenter une simple méthode
    - p.ex. action à faire après un clic sur un bouton (cf. exemple précédent)

- Les expressions Lambda: (*parameterList*) **->** { /* ⋯ */ }
  - (*parameterList*)

(ActionEvent evt, ⋯) ou evt (un seul paramètre)

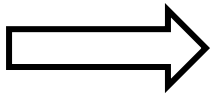
  - { /*… */ } : le corps (*body*)

```
{ ⋯
   clicInstall( id);
   ⋯
}
```

ou

```
clicInstall( id )
```

(une seule expression)



ou

```
( ActionEvent evt ) -> { clicInstall( id ) }
```

```
evt -> clicInstall( id )
```

```
import java.awt.event.*;
// …
public class MainWindow extends JFrame{
    …
    public MainWindow() {
        int id = 0;
        // …
        installButton.addActionListener(
                evt ->  clicInstall( id )
        );
        // ⋯
    }
    private void clicInstall( int id ) {
        // ⋯
    }
}
```

Bonnes pratiques : conventions de nommage

- Conventions définies initialement par Sun pour
  - faciliter la maintenance
    - 80% du temps de vie d'un logiciel et logiciel rarement maintenu par son auteur initial
  - faciliter la lecture du code (et donc son débogage)
  - faciliter la commercialisation et la diffusion

Packages	noms séparés par des ".", tout en minuscule, au début nom de domaine	*com.sun.eng* *nc.univ-nc.poo*
Classes (et Interfaces)	nom dont la 1ère lettre est en majuscule, mélange de minuscule et de majuscule avec la première lettre de chaque mot en majuscule	*class HelloWorld* *class Raster*
Méthodes	verbes dont la 1ère lettre est en minuscule, mélange de minuscule et de majuscule avec la première lettre de chaque mot en majuscule	*run()* *getBackground()*
Variables	noms dont la 1ère lettre est en minuscule, mélange de minuscule et de majuscule avec la première lettre de chaque mot en majuscule (*utiliser i, j, k, m, n* pour des variables temporaires entières et *c, d, e* pour des caractères)	*int i;* *char c;* *float myWidth;*
Constantes	noms tout en majuscule séparés par des "_"	*static final int MIN_WIDTH = 4;*

- De manière générale,
  - Utiliser des mots simples et descriptifs
  - Eviter les accents et les caractères spéciaux (p.ex. $, *, ...)

Bonnes pratiques : documenter le code (Javadoc)

- Facilite la compréhension et donc l'utilisation, le débogage et l'évolution des programmes et des librairies
  - commenter le plus possible
  - commenter chaque déclaration de variable, de méthode et de classe

- Programme **javadoc**
  - 3e programme essentiel fourni avec le SDK (java, javac)
  - Génère automatiquement la documentation au format de Sun (API) à partir des commentaires Javadoc
    - documentation sous forme de pages HTML, PDF, etc
    - liens hypertextes pour la navigation

➢ Un outil fondamental pour bien programmer en Java

Bonnes pratiques : documenter le code (Javadoc)

- Commentaire Javadoc

```
/**
* pour l'utilisation de Javadoc
* à réserver pour la documentation automatique
* avec javadoc
*/
```

- Utilisation de tags (p.ex. *@param* ou *@return*) pour documenter plus précisément le code source
  - impact sur la documentation générée : création de liens hypertextes, de tableaux de variables, ...

- Exemple :

```
/**
* Addition de deux entiers.
* @param a Le premier entier.
* @param b Le second entier.
* @return La somme de a et de b.
*/
public int addition (int a, int b) {
                return a + b;
}
```

Bonnes pratiques : documenter le code (Javadoc)

- Quelques tags Javadoc

Tag	***Utilisé où***	***Description***
@author *name*	Classes et Interfaces	Nom du développeur
@version *description*	Classes et Interfaces	Version du code source
@deprecated	Classes et Interfaces	Code déprécié. Certains IDEs créent un avertissement à la compilation si le code est appelée.
@param *name description*	Méthodes	Paramètre de méthode (un tag pour chaque paramètre de la méthode)
@return *description*	Méthodes	Valeur de retour (uniquement si une fonction)
@throws *name description*	Méthodes	Exception (i.e. erreur) lancée par la méthode
@see *Classname*	Partout	Lien vers une autre classe
@see *Classname*#member	Partout	Lien vers un membre (méthode ou champ) d'une classe

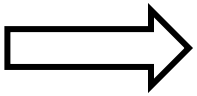
- Ordre d'apparition des tags en général : @author, @version, @param, @return, @throws, @see, @since, @deprecated

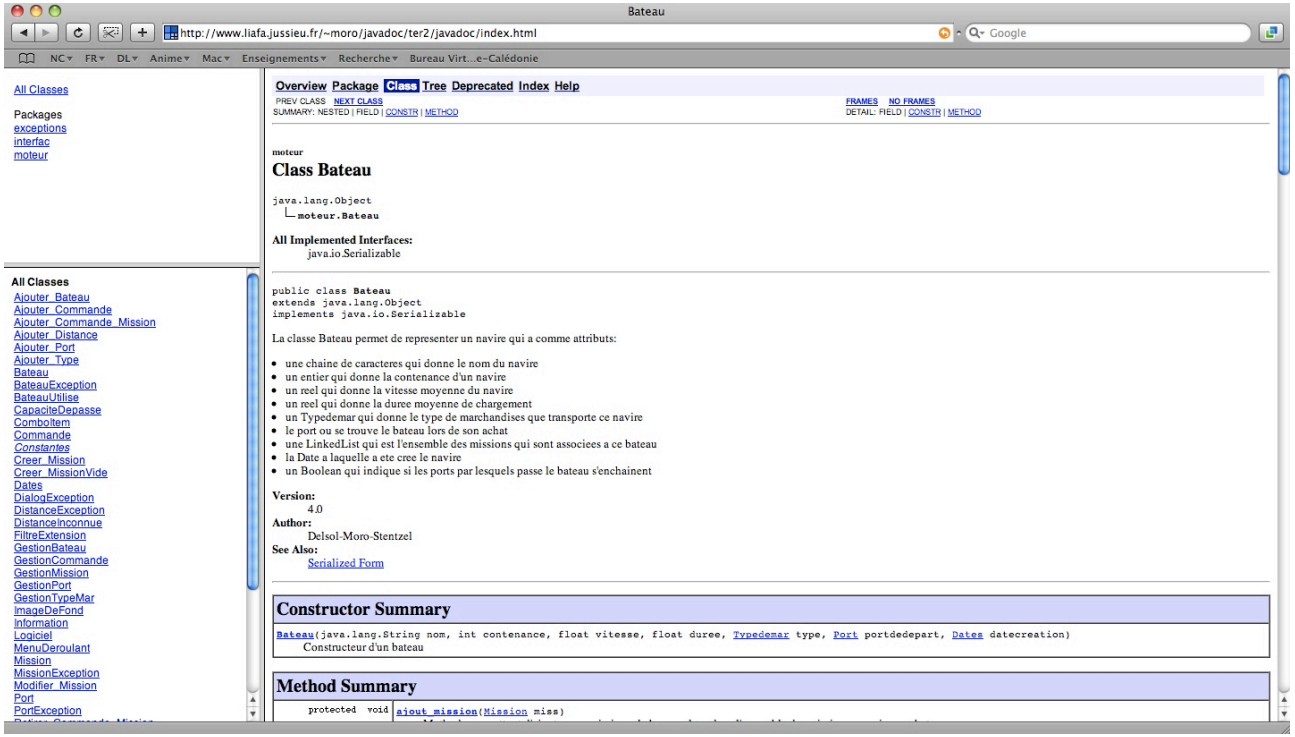
Bonnes pratiques : documenter le code (Javadoc)

- Construction de la documentation via la commande

  javadoc −d *myPath myfile*

  - avec *myPath* répertoire de destination et *myFile* le nom des fichiers Java
  - p.ex. *javadoc -author –d ~/TP/ *.java*





Exercice: Intégration du nombre de livres empruntables et des durées d'emprunt

- Intégrer deux attributs permettant de stocker le nombre de documents empruntables au maximum (commun à toutes les bibliothèques) et le nombre de jours maximum pendant lequel un document peut être empruntés (spécifique à chaque bibliothèque).
  - Initialiser ces attributs dans les constructeurs implémentés (réutiliser le code du constructeur par défaut dans le second)
  - Implémenter deux méthodes (statique) permettant de mettre à jours ces attributs

- Ajouter la documentation JavaDoc aux classes *Client* et *Document* (et à leurs attributs et méthodes), puis générer la documentation technique associée à partir d'Eclipse
  - *Project* > *Generate Javadoc …*