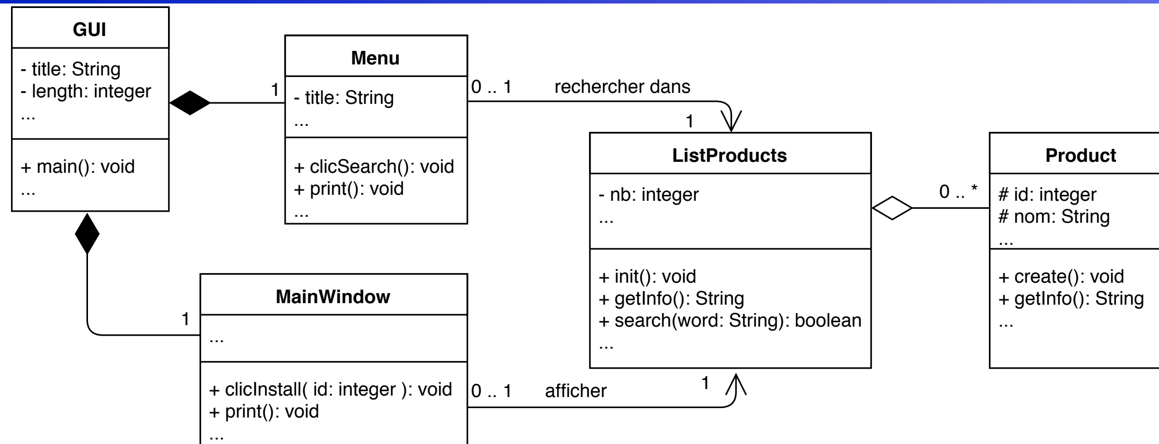
Plan

- 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

Agrégation et composition en Java

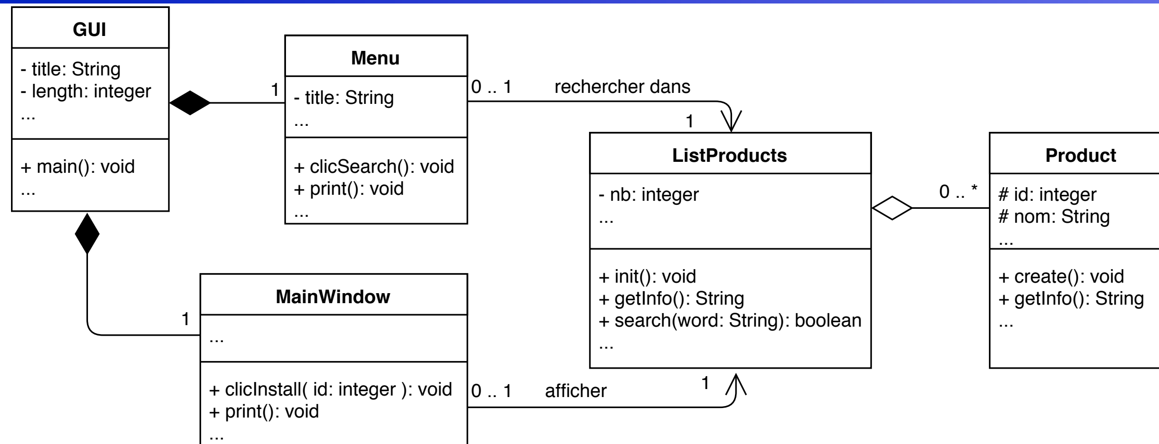


- Agrégation = relation "a un" (*"has-a"*)
  - Un attribut dans la classe "agrégeant"
    - éventuellement un tableau ou une "collection"
  - Entités indépendantes
    - stockage d'une référence vers un objet externe
    - un paramètre du constructeur
- Composition = relation "est composé de" (*"part-of"*)
  - Un attribut dans la classe "composée de"
    - éventuellement un tableau ou une "collection"
  - Parties ne peuvent exister indépendamment
    - créé et initialisé dans le constructeur

```
import java.util.*;

public class ListProducts {

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

    public void addProduct( Product prod ){
          products.add( prod );
    }
    // …
}
```

```
public class GUI {

    private String title = "Google Play" ;
    private int length = 240 ;
    private final Menu prodMenu ;
    private final MainWindow prodWindow ;
    // …
    public GUI() {
          prodMenu = new Menu() ;
          prodWindow = new MainWindow();
    }
    // …
}
```

Associations en Java




```
import java.awt.event.*;
// …
public class MainWindow extends JFrame{
    ListProducts myList ;
    // …
    public MainWindow() {
        // …
    }
    private void print() {
        // …
    }
    // …
}
```

- Toute relation entre deux classes
  - Sémantiques très variées
    - p.ex. "rechercher dans" ou "afficher"
  - Relation unidirectionnelle ou bidirectionnelle

- Un attribut ou plusieurs dans une classe ou les deux
- Une généralisation de la notion d'agrégation et de composition

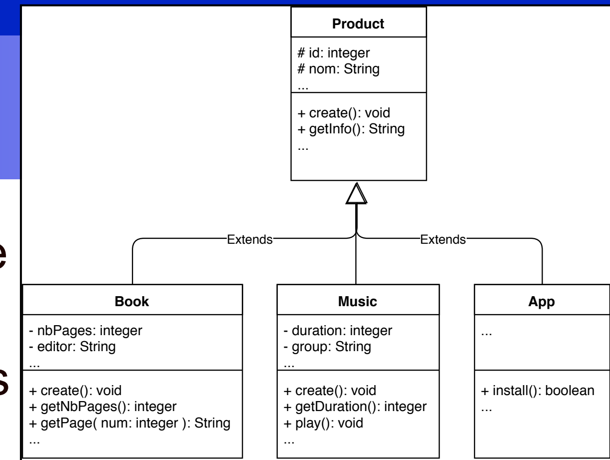
Exercice: Création de la classe *FicheEmprunt* associée à des documents et un client

- Compléter (et tester) la classe *FicheEmprunt* avec les attributs et méthodes suivants:
  - attribut *listeEmprunts: ArrayList<Document>* afin de représenter la composition entre *FicheEmprunt* et *Document*
  - attribut *client: Client* afin de représenter l'association entre *FicheEmprunt* et *Client*
  - constructeur par défaut: *FicheEmprunt ()*
    - initialiser cet attribut dans le constructeur
  - constructeur permettant d'initialiser les attributs: *FicheEmprunt(client: Client)*
  - méthode permettant de comparer si deux fiches sont égales: *equals(fiche: FicheEmprunt): boolean*
  - méthode *ajouterEmprunt(doc: Document): boolean* permettant d'ajouter un document dans la fiche d'emprunts
    - utiliser la méthode *add* de la classe *ArrayList*
  - méthode retournant une description de fiche: *toString():String*
- Modifier la classe *Document* afin d'ajouter un attribut *fiche: FicheEmprunt* stockant une référence à une fiche emprunt si celui est emprunté. Puis, mettre à jours cet attribut lorsqu'un document est emprunté/rendu.
  - modifier la méthode *ajouterEmprunt* de *FicheEmprunt* et utiliser la méthode *setFicheEmprunt(fiche: FicheEmprunt)* de Document pour modifier cet attribut

Exercice: Gestion de la liste des documents et des clients

- Stocker la liste des documents dans la classe *Bibliotheque*
  - Ajouter/modifier un attribut *listeDocuments: ArrayList<Document>* afin de représenter la composition entre *Bibliotheque* et *Document*
  - Initialiser cet attribut dans les constructeurs de *Bibliotheque*
  - Implémenter (et tester) la méthode *ajouterDocument(doc: Document): boolean* permettant d'ajouter un document dans une bibliothèque
    - utiliser la méthode *add* de la classe *ArrayList*
  - Implémenter (et tester) la méthode *supprDocument(code: String): boolean* permettant de supprimer un document d'une bibliothèque
    - utiliser la méthode *remove* de la classe *ArrayList*
  - Tester les méthodes créées par l'intermédiaire d'une classe *BibliothequeTest*
- Faire de même pour la liste des clients
- Gérer les emprunts (toujours dans la classe *Bibliotheque*)
  - Implémenter (et tester) la méthode *emprunterDocument(code: String[], idClient: int): boolean* permettant de créer une fiche d'emprunt associant une liste de documents et un client
  - Implémenter (et tester) la méthode *rendreDocument(code: String)* permettant de rendre disponible un document emprunté

L'héritage en Java



- Réutiliser des attributs et des méthodes d'une classe existante sans avoir à tout réécrire (et re-débugger)
  - Classes filles "héritent" des attributs et méthodes publiques et protégés de leur classe mère
    - les constructeurs de la classe mère ne sont pas hérités par les classes filles, mais ils peuvent être invoqués à partir de celles-ci
  - Représente une relation "est un" (*"is-a"*)

- Définition: [modifier] **class** *SubClass* **extends** *SuperClass*
  - une seule classe mère possible

```
public class Product {
    protected int id;
    protected String nom;
    // …
    public void create() {
        // …
    }

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

```
public class Book extends Product {
    protected int nbPages = 0;
    protected String editor ;
    // …
    public int getNbPages() {
        return nbPages ;
    }
    // ….
    public String getInfo() {
        return id+"  : "+nom+", Editor :"+editor ;
    }
    // …
}
```

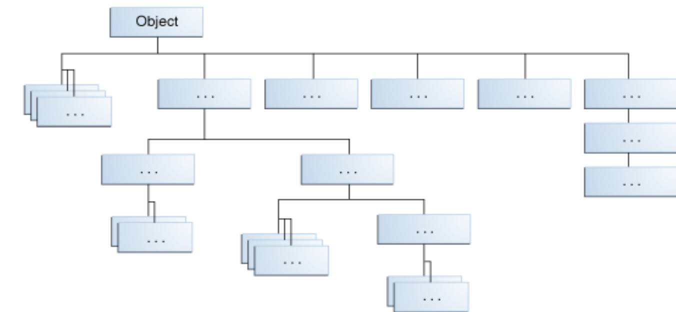
redéfinition (*overriding*)

L'héritage en Java

```
public class Book extends Product {
    protected int nbPages = 0;
    protected String editor ;
    // …
    public Book( int id, String nom, String editor ) {
        super( id, nom);
        this.editor = new String( editor );
    }
    // ….
    public String getInfo() {
        return super.getInfo()+", Editor :"+editor ;
    }
    // …
}
```

- Utiliser dans une classe fille le mot-clé **super**
  - pour invoquer les constructeurs de la classe mère
  - pour invoquer une méthode de la classe mère surchargée dans la classe fille

- La hiérarchie de classes de l'API de Java
  - Classe **Object**: classe mère de toutes les classes Java (même les nôtres)



- Transtypage (*casting*) d'objets
  - Les objets instanciant les classes filles peuvent être utilisés partout où la classe mère est attendue (l'inverse n'est pas vrai)

```
Product newProd = new Book() ;
Object obj = new Book() ;
newProd = obj ; // erreur à la compilation
Book mybook = newProd ; // erreur à la compilation
```

```
Product newProd = new Book() ;
Object obj = new Book() ;
newProd = (Book) obj ;
Book mybook = (Book) newProd ;
```

Exercice: Gérer différents types de documents (livres, CD audio et vidéos)

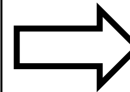
- Implémenter l'héritage entre documents, livres, CD audio et vidéos. Pour cela, redéfinir et tester les méthodes suivantes dans chacune des sous-classes :
  - les constructeurs
  - la méthode *toString*
  - la méthode *equals*

- Modifier la méthode *estEmpruntable()* de *Video* afin de retourner tout le temps faux pour les vidéos

Le polymorphisme dynamique en Java

- Possibilité de redéfinir dans les classes filles des méthodes de la classe mère
  - A l'exécution, appel de la méthode appropriée pour l'objet référencé (fait par la JVM)

```
Product[] arrayProducts= new Product[3] ;

arrayProducts[0] = new App( 0, "Android");
arrayProducts[1] = new Book( 1, "Fondation", "Hachette") ;
arrayProducts[2] = new Music( 2, "Black Album", );

for( int i=0; i< 3; i++) {
   System.out.println( arrayProducts[i].getInfo() );
}
```



Invocation des méthode **getInfo**() des classes **App**, **Book** et **Music**

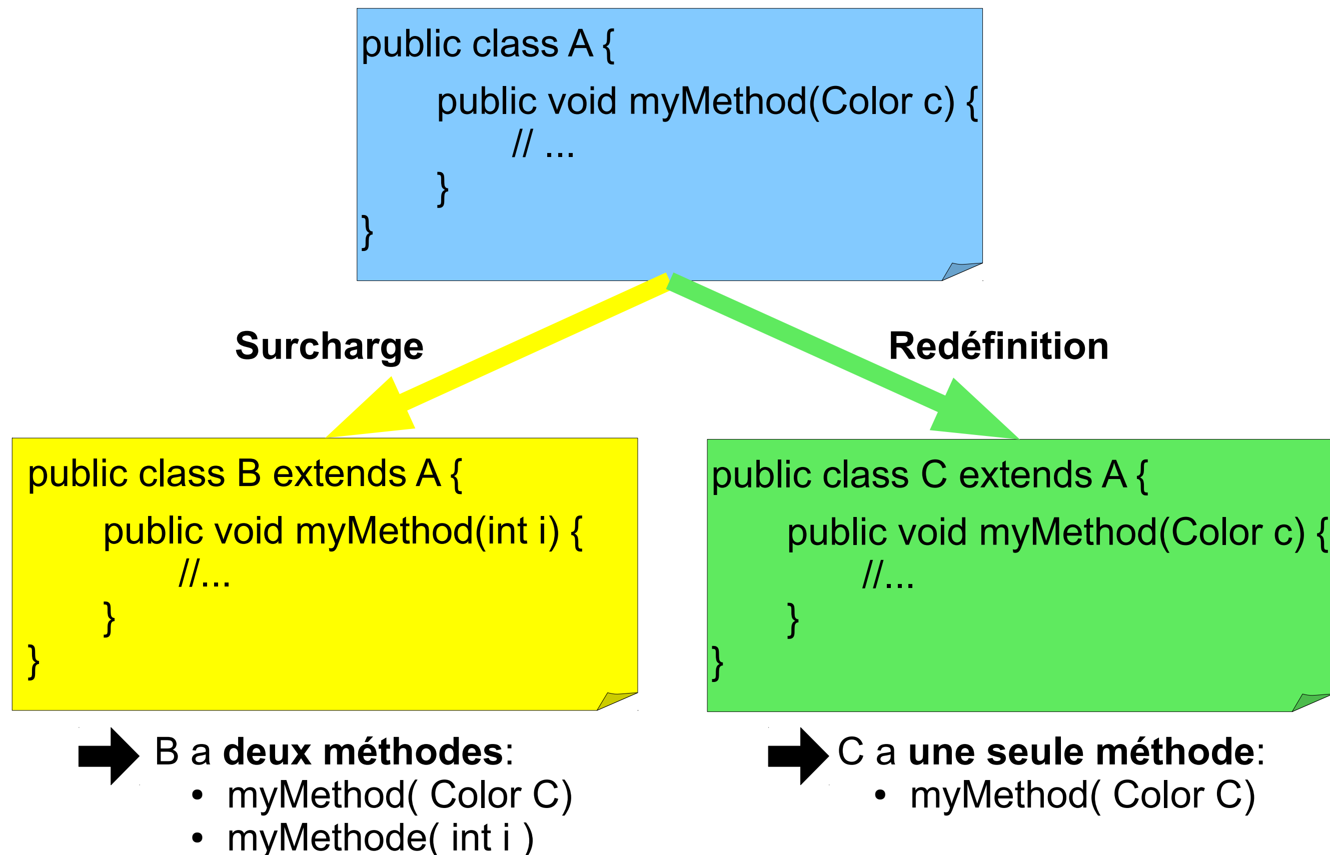
```
0 : Android
1 : Fondation, Editor : Hachette
2 : Black Album (nb tracks : 12)
```

- Attention: lorsqu'un objet est "sur-classé", il est vu comme un objet de la classe de la variable utilisée pour le désigner
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe de la variable

```
Product myProd = new Book( 1, "Fondation", "Hachette") ;
int nbPages = myProd.getNbPages() ; // erreur à la compilation
Book myBook = (Book) myProd ;
int nbPages2 = myBook.getNbPages() ; // OK
```

Le polymorphisme statique vs dynamique

- Ne pas confondre redéfinition et surcharge !



Exercice: Gérer différents types de documents (livres, CD audio et vidéos)

- Exploiter le polymorphisme pour afficher la liste des documents quelque soit leur type (livre, cd audio ou vidéo)
  - Implémenter la méthode *toString(): String* dans la classe *Bibliotheque*. Cette méthode retournera la liste des documents contenus dans la bibliothèque (via l'attribut *listeDocuments*), en intégrant les informations propres à chaque type de documents.

- Développer dans la méthode *main* de *Bibliotheque* une interface utilisateur ("textuelle", dans le terminal). Elle devra permettre dans un premier temps de
  - saisir les différents types de documents
    - saisie au clavier le type du document à ajouter et les informations correspondantes
    - création/instanciation d'un objet de la classe correspondante
    - ajout dans la liste des documents
  - afficher les documents saisis
    - utiliser la méthode *toString()*
  - sélectionner des documents (empruntables) et de créer les fiches d'emprunts associées
    - utiliser la méthode *emprunterDocument(code: String[], idClient: int): boolean*

Ecrire des classes et méthodes finales

- Limiter l'héritage et le polymorphisme
  - Une méthode finale ne peut être redéfinie dans les classes filles
    - méthodes privées implicitement finales
    - conseillé de déclarer en finales les méthodes appelées dans les constructeurs
    - optimisation des méthodes finales par le compilateur
  - Une classe finale ne peut avoir de classes filles (interdit tout héritage)

- Déclaration:

[modifier] [**static**] **final** *returnsType methodName*(*paremeterList*) { ··· }

[modifier] **final** **class** *MyClass* [**extends** *MySuperClass*] [**implements** *YourInterfaces*]

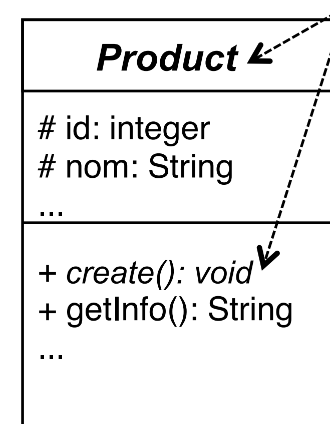
```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    // ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    // ...
}
```

Les classes et méthodes abstraites

- Classe abstraite (**abstract**) = une classe qui ne peut être instanciée (mais peut être héritée)
  - Pas de création d'objets de cette classe
  - Permet de partager/factoriser du code commun à plusieurs classes sémantiquement reliées
    - des attributs et/ou des méthodes (pas forcément abstraites)

```
public abstract class Product {
    protected int id;
    protected String nom;
    // …
    public abstract void create() ;

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

- Méthode abstraite = une méthode déclarée mais sans implémentation
  - Seulement une signature de méthode
  - Permet de définir des fonctionnalités incomplètes qui devront être implémentées dans les classes filles

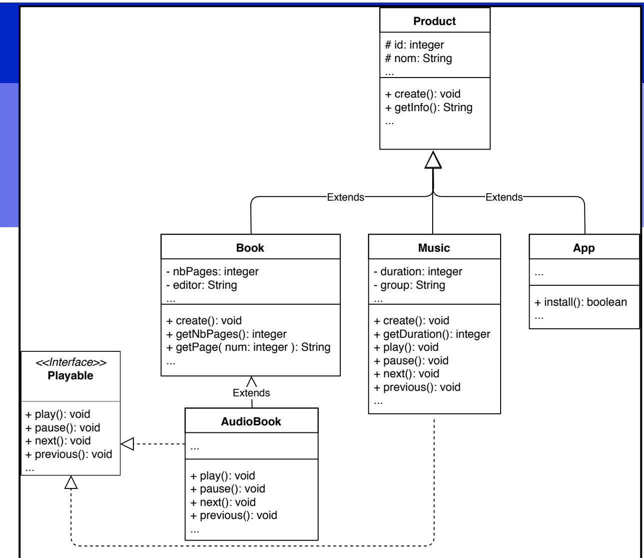


Exercice: Interdire la création de documents

- Transformer la classe *Document* en classe abstraite afin d'en interdire l'instanciation

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Les interfaces en Java



- Un *contrat*, un ensemble de fonctionnalités, devant être implémenté par des classes
  - Uniquement des constantes, des signatures de méthodes, des méthodes par défaut, des méthodes statiques, et des types imbriqués
- Définition: [modifier] **interface** *MyInterface* [**extends** *InterfaceList*]
  - Toutes les méthodes sont implicitement "publiques"
- Implémentation: [modifier] **class** *YourClass* **implements** *MyInterface* [*InterfaceList2*]

```
import java.time.*;
import java.sound.*;

public interface Playable {

  void play() ;
  void pause() ;
  void next() ;
  void previous() ;

  private String getTimeZone() {
        TimeZone tz = TimeZone.getDefault();
        return tz.getDisplayName(false,
                           TimeZone.SHORT);
  }

 default void increaseVolume() {
       // …
  }
}
```

```
public class AudioBook extends Book implements Playable {
 // …
  void play() {
    // …
  }

  void pause() {
    // …
  }

  void next() {
    // …
  }

  void previous() {
    // …
  }
}
```

Les interfaces en Java

- Une classe peut implémenter plusieurs interfaces

- Possibilité d'utiliser une interface comme un type
  - Type d'une variable ou type d'un paramètre d'une méthode
  - Objet "affecté" doit être une instance d'une classe implémentant l'interface

```
public void playAll( ArrayList arrayPlayable ) {
    for( int i = 0; i < arrayPlayable.length ; i++ ) {
        Playable current = (Playable)( arrayPlayable.get(i) ) ;
        current.play() ;
    }
}
```

```
...
ArrayList all = new ArrayList();
all.add( new Audiobook("Le petit Prince") );
all.add( new Music("GoT soundtrack") );
...
```

- Evolution des interfaces
  - Attention: ajout d'une nouvelle méthode dans une interface ➔ toutes les classes l'implémentant ne marche plus tant que cette nouvelle méthode n'a pas été implémentée dans chacune d'entre elles
    - importance d'anticiper le plus possible toutes les utilisations de l'interface dès le départ
  - Des solutions:
    - définir une autre interface étendant la première
    - définir un comportement par défaut pour le nouvelle méthode

Exercice: Généraliser la notion de documents empruntables

- Ajouter l'interface *Empruntable.* Cette interface contient la méthode *setFicheEmprunt(fiche: FicheEmprunt).*
  - supprimer cette méthode de la classe *Document* afin d'éviter une erreur de redéfinition
  - implémenter cette méthode dans les classes *Livre* et *Audio*
  - modifier le code de la méthode *ajouterEmprunt* de *FicheEmprunt* afin qu'elle puisse utiliser cette méthode spécifique aux documents empruntables
    - faire un transtypage explicite en *Empruntable* du document référencé avant d'utiliser la méthode *setFicheEmprunt*

- Modifier les classes *Livre* et *Audio* afin qu'elles implémentent l'interface *Empruntable*

- Modifier le code de votre classe *FicheEmprunt* afin de contraindre les emprunts aux documents empruntables.
  - Modifier l'attribut *listeEmprunts: ArrayList<Document>* en *listeEmprunts: ArrayList<Empruntable>*

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

UNIVERSITÉ de la NOUVELLE-CALÉDONIE