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 packages en Java

- Organisation du code visant à grouper des types (classes, interfaces, etc) liés
  - Similaire à la notion de répertoire dans les systèmes d'exploitation
  - Utilisation et recherche de fonctionnalités plus faciles, évite les conflits de nom, facilite le contrôle des accès, etc
  - API Java composée de packages: java.io, java.util, java.awt.event, etc

- Créer un package: **package** *myPackage***;**
  - Convention de nommage: nom de domaine inversé en minuscule suivi du nom du package (précédé si nécessaire par le nom du projet)

```
package nc.unc.products;
public class Product {
    // …
}
```

Product.java

```
package nc.unc.products;
public class Book
            extends Product {
    // …
}
```

Book.java

```
package nc.unc.products;
public class App
            extends Product {
    // …
}
```

App.java

```
package nc.unc.products;
public interface Playable {
    // …
}
```

Playable.java

```
package nc.unc.products;
public class Music extends Product implements Palayable {
    // …
}
```

Music.java

```
package nc.unc.products;
public class AudioBook extends Book implements Palayable {
    // …
}
```

AudioBook.java

Utiliser un package

- Accéder à un membre (classe, interface, etc) d'un package à l'extérieur de celui-ci
  - Utiliser son nom complet

```
nc.unc.products.Music myMusic = new nc.unc.products.Music() ;
```

  - Importer avec la commande **import**
    - le membre du package visé: **import** *myPackage.myType* **;**
    - le package en entier: **import** *myPackage*.*;

```
import nc.unc.products.Music;
Music myMusic = new Music() ;
```

```
import nc.unc.products.*;
Music myMusic = new Music() ;
```

- Pas de hiérarchie de packages
  - Ex. package java.awt (*Abstract Window Toolkit*), package java.awt.color, java.awt.font, etc
  - ➢ Des packages indépendants, aucune inclusion
    - *import java.awt.** n'importe pas *java.awt.font* p.ex.
    - il faut inclure les deux

```
import java.awt.*;
import java.awt.font.*;
//…
```

- Attention aux conflits de noms
  - Si des packages importés ont des membres avec des noms identiques
  - ➢ Préfixer le membre par le nom du package

```
graphics.Rectangle rect;
```

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Exercice: Réorganisation du code en packages

- Réorganiser le code de l'application de gestion des prêts en créant trois packages:
  - un package autour des documents,
  - un package autour des clients,
  - un package autour de la gestion de la bibliothèque.

Organisation des codes sources et des fichiers class

- Organisation du code source
  - Mettre le code source d'une classe, interface ou énumération dans un fichier texte dont le nom est celui du type en question et ajouter l'extension *.java*
    - p.ex. fichier Product.java, Book.java, etc
  - Puis mettre ce fichier source dans répertoire dont le nom correspond au package auquel il appartient
    - p.ex. fichier *Product.java, Book.java*, etc dans le répertoire *…../nc/unc/products/*

- Organisation des fichiers class (fichiers "compilés")
  - Par défaut, même répertoire que le code source
  - Possibilité de séparer fichiers sources et fichiers class dans des répertoires différents
    - intérêt: masquer le code source aux autres développeurs
    - p.ex. ***path_one/sources****/nc/unc/products/Product.java* et ***path_two/classes****/nc/unc/products/Product.class*
    - ajouter ce répertoire de classes (*class path*) dans les variables d'environnements

      *CLASSPATH=/****path_two/classes****; export CLASSPATH* (UNIX)

      *set CLASSPATH=****path_two\classes*** (WINDOWS)

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Plan

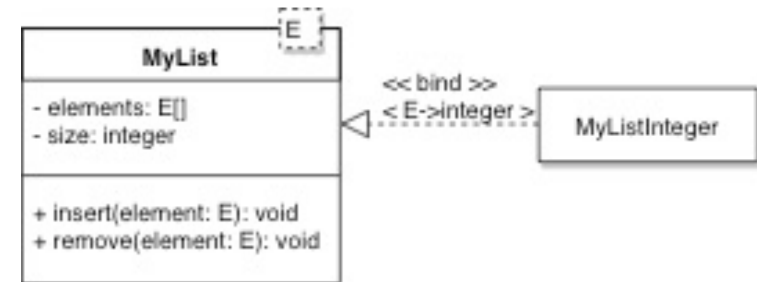
- Introduction au Java
  - Généralités
  - Syntaxe de base

- 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

La généricité en Java

- Paramétrer les types utilisés dans une classe, une interface ou une méthode
- Avantages:
  - Une vérification plus forte des types à la compilation
    - plus simple à débugger
  - Plus besoin de transtypages (*cast*)
  - Développement de codes génériques
    - optimisés, plus sûrs et plus simple à lire




```
public class MyList<E> {
    private E[] elements;
    private int size = 0;

    public void insert(E element){
        // …
    }
    public void remove(E element){
        // …
    }
}
```

- Types génériques (classes ou interfaces génériques)
  - Déclaration: **class** *MyClass*<*T1, T2, …,Tn*>**{ … }**
    - conventions de nommage: E – élément, K- clé, N – nombre, T – type, V- valeur, …
  - Invocation: *MyClass*<*myType1, …, myTypeN*> *myVariable*;

```
MyList<Product> listproducts = new MyList<Product>();
```

```
MyList<Product> listproducts = new MyList<>(); // type inference
```

La généricité en Java

- Méthodes génériques
  - Déclaration: **<***T1, T2, ⋯,Tn***>** *returnsType methodName***(** *paremeterList* **) { ⋯ }**
  - Invocation: *myObject*.**<***myType1, ⋯, myTypeN***>***methodName***(***paremeterList***);**

```
public class Util {
   public static <K, V> boolean compare( Pair<K, V> p1,
                                          Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
   }
}
```

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);

boolean same2 = Util.compare(p1, p2); // OK aussi
                                      // type inference
```

- Les types paramétrés bornés (*bounded type parameters*)
  - Restreindre les types passés en argument en utilisant le mot-clé **extends**
    - seules les classes ou interfaces dérivées peuvent être utilisées
  - Possibilité d'avoir plusieurs bornes: **<***T* **extends** *B1* **&** *B2* **&** ... *BN***>**

```
public static <T extends Comparable<T>> int countGreaterThan(Collection<T> aCollec,
                                                              T elem) {
   int count = 0;
   for (T e : aCollec)
      if (e.compareTo(elem) > 0)  ++count;
   return count;
}
```

(pour chaque élément d'une collection ou d'un tableau)

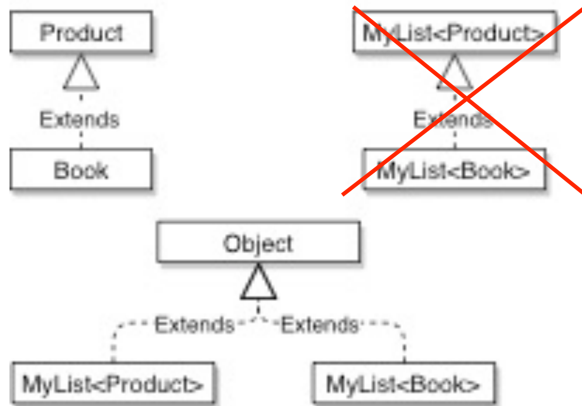
```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

package java.lang

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Généricité et héritage

- Erreur communément faite




```
Product myProd ;
Book myBook = new Book();
myProd = myBook ; // OK

MyList<Product> listProducts = new MyList<Product>();
listProducts.add( new Book() ); // OK
listProducts.add( new Product() ); // OK

MyList<Book> listBooks = new MyList<Books>();
listProducts = listBooks; // error
```

- Créer un sous-type générique
  - id. classes et interfaces classiques ➔ **extends** et **implements**

```
public interface List<E>  extends Collection<E> {
        // …
}
```

```
public class MyList<E>  implements List<E> {
        // …
}
```

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

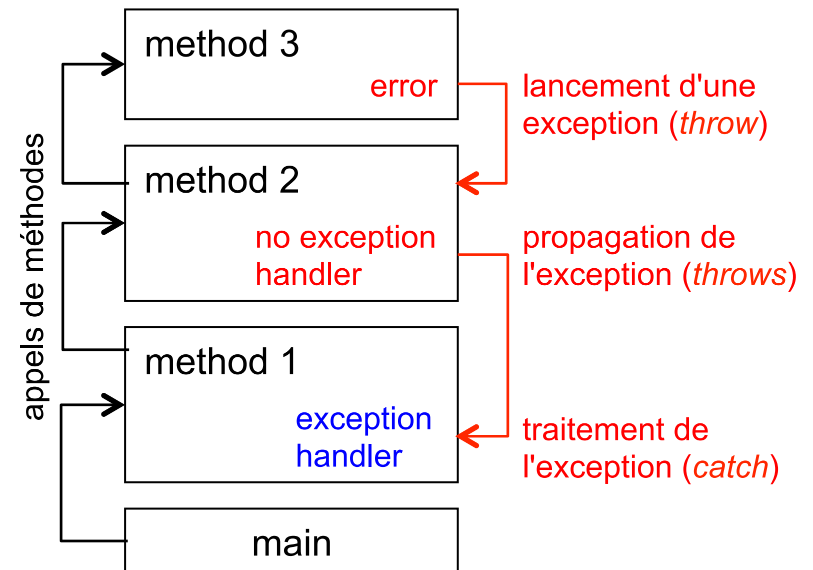
Plan

- 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 exceptions

- Des "évènements exceptionnels" = des évènements apparaissant durant l'exécution du programme et interrompant le flot normal des instructions
  - Typiquement une erreur à l'exécution (*runtime error*)

- Gestion des exceptions en Java
  - Séparation du traitement des erreurs du code normal
  - Propagation automatique des erreurs en remontant la pile des appels
  - Regroupement et différentiation des erreurs par types

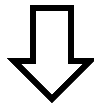


- Attraper ou spécifier ("*catch or specify*")
  - Toute méthode susceptible de lever une exception **doit**
    - soit l'attraper et la traiter (***try*** ... ***catch***...),
    - soit déclarer explicitement que l'exception peut être lancée pour qu'elle puisse être propagée et traitée ailleurs (***throws***)

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Exemples de programmes lançant une exception

```
public class Product {
    protected int id = 0;
    protected String nom;
    // …
    public Product(String id, String nom) {
        this.id = Integer.parseInt( id );
        this.nom = new String( nom );
    }
    // …
    public static void main( String[] args ) {
        Product testProd = new Product( args[0], args[1] ) ;
    }
}
```

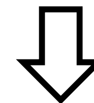


```
> java Product zero bob
> Exception in thread "main"
java.lang.NumberFormatException: For input string: "zero"
    at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at Product.<init>(Product.java:7)
    at Product.main(Product.java:12)
```

```
import java.util.*;

public class ListProducts {

    private int nb = 0 ;
    private ArrayList products;
    // …
    public Product productAt( int index ) {
        return (Product) products.get( index ) ;
    }
    // …
    public static void main( String[] args ){
        ListProducts testList = new  ListProducts();
        Product prod5 = testList.productAt( 5 );
    }
}
```



```
> java ListProducts
> Exception in thread "main" java.lang.NullPointerException
    at ListProducts.productAt(ListProducts.java:9)
    at ListProducts.main(ListProducts.java:14)
```

Les trois catégories d'exceptions

- Les **exceptions contrôlées** (*checked exceptions*)
  - Conditions exceptionnelles qu'une application doit anticiper et traiter
    - utilisateur saisi un mauvais nom de fichier qui déclenche une exception *java.io.FileNotFoundException*
    - l'application doit attraper cette exception et notifier l'utilisateur de l'erreur commise
  - Erreur à la compilation si "*catch or specify*" non mis en place
  - Toutes les exceptions sont de ce type sauf celles héritant des classes *Error* et *RunTimeException*

- Les **erreurs** (*errors*)
  - Conditions exceptionnelles extérieures à l'application et difficilement anticipables
    - un mauvais fonctionnement du disque lors d'une lecture de fichier déclenchera une exception *java.io.IOError*
  - Pas obligatoire de mettre en place le "*catch or specify*"
  - Les exceptions dérivées de *Error*

- Les **exceptions levées à l'exécution** (*runtime exception*)
  - Conditions exceptionnelles internes à l'application et difficilement anticipables
    - un bug de programmation qui déclenche une exception *NullPointerException*
  - Pas obligatoire de mettre en place le "*catch or specify*"
  - Les exceptions dérivées de *RunTimeException*

Gérer les exceptions

- Exemple de code ne compilant pas car ne respectant pas le "catch or specify"

```
import java.io.*;
public class Product {
    // …
    public void printFile() {
        FileWriter file = new FileWriter(id+".txt") ;
        file.write( id+" : "+nom );
        file.close() ;
    }
}
```

FileWriter

```
public FileWriter(String fileName)
           throws IOException
```

Constructs a FileWriter object given a file name.

Parameters:

`fileName - String The system-dependent filename.`

Throws:

`IOException - if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason`

Solution 1: gestion active avec **try/catch**

```
import java.io.*;
public class Product {
    // …
    public void printFile() {
        FileWriter file ;
        try {
            file = new FileWriter(id+".txt") ;
            file.write( id+" : "+nom );
            file.close() ;
        }
        catch( IOException e ) {
            System.err.println("Erreur lors de l'écriture dans le fichier");
        }
    }
}
```

Solution 2: gestion passive avec **throws**

```
import java.io.*;
public class Product {
    // …
    public void printFile() throws IOException {
        FileWriter file = new FileWriter(id+".txt") ;
        file.write( id+" : "+nom );
        file.close() ;
    }
}
```

Quelle solution ?
Arrêter avec **catch** les exceptions que l'on peut traiter et propager avec **throws** les autres

Attraper et traiter les exceptions

- Bloc **try { ⋯ } catch(** *ExceptionType name* **){ ⋯ }**
  - *try{ … }* = une ou plusieurs lignes pouvant lancer des exceptions
  - *catch(…){ … }* = code gérant l'exception passée en argument
    - plusieurs possibles: traitement fait par le premier qui "attrape"
    - attention à l'ordre des *catch* et à l'héritage: toujours du plus spécifique au plus général
    - ne pas tout attraper …

```
import java.util.*;

public class ListProducts {
   // ⋯
   public Product productAt( int index ) {
         Product prod ;
         try {
            prod = (Product) products.get( index ) ;
         }
         catch( NullPointerException e) {
            products = new ArrayList() ;
         }
         catch( IndexOutOfBoundsException e ) {
            System.err.printl("Bad product index");
            e.printStackTrace();
         }
         return prod ;
   }
}
```

```
import java.util.*;

public class ListProducts {
   // ⋯
   public Product productAt( int index ) {
         Product prod ;
         try {
            prod = (Product) products.get( index ) ;
         }
         catch( NullPointerException|IndexOutOfBoundsException  e) {
            System.err.println("Caught: " + e.getMessage());
         }
         return prod ;
   }
}
```

get

`public E get(int index)`

Returns the element at the specified position in this list.

Specified by:
`get` in interface `List<E>`

Specified by:
`get` in class `AbstractList<E>`

Parameters:
`index` - index of the element to return

Returns:
the element at the specified position in this list

Throws:
`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

Attraper et traiter les exceptions

- Bloc **finally{ ⋯ }**
  - Instructions toujours exécutées (même si une exception est levée)
    - même si un **return** dans un bloc **catch** le précédent
  - Très utile pour libérer des ressources utilisées dans le bloc **try**

- Instruction **try** avec des ressources (*try-with-resources*): **try(⋯)**{ ⋯} catch( ){ ⋯ }
  - Permet de libérer automatiquement toute ressource implémentant *java.lang.AutoCloseable* et *java.io.Closeable*
  - Bloc avec plusieurs instructions possibles en argument de **try**

```
import java.io.*;
public class Product {
    // ⋯
    public void printFile() {
        FileWriter file ;
        try {
            file = new FileWriter(id+".txt") ;
            file.write( id+" : "+nom );
        }
        catch( IOException e ) {
            System.out.println("Error reading file");
            e.printStackTrace();
        }
        finally {
            try {
              if( file != null ) file.close();
            }
            catch( IOException e ) {
               System.err.println("Error closing file");
            }
        }
    }
}
```

```
try ( FileWriter file = new FileWriter(id+".txt"); ) {
    file.write( id+" : "+nom );
}
catch( IOException e ) {
    System.out.println("Error reading file");
    e.printStackTrace();
}
```

UNIVERSITÉ de la NOUVELLE-CALÉDONIE

Attraper et traiter les exceptions

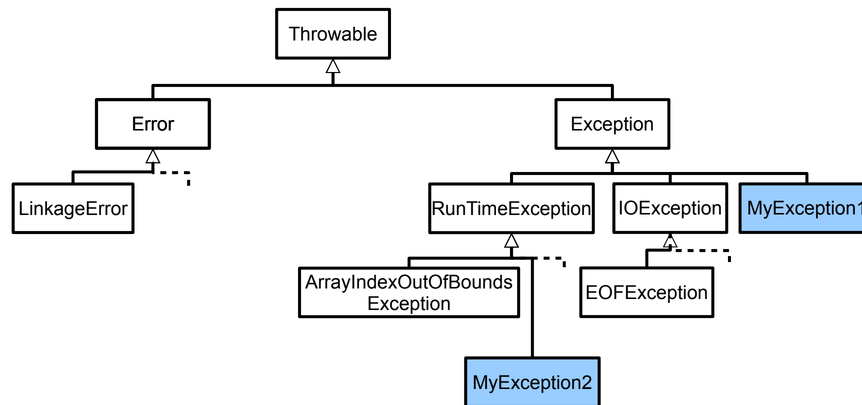
- Possibilité d'enregistrer les erreurs dans un fichier log
  - Enregistrer l'historique des erreurs plutôt que les afficher dans le terminal
  - Utiliser le package **java.util.logging**

```
import java.io.*;
import java.util.*;

public class Product {
    // …
    public void printFile() {
        FileWriter file ;
        try( file = new FileWriter(id+".txt") ; ) {
            file.write( id+" : "+nom );
        }
        catch( IOException e ) {
            Logger logger = Logger.getLogger("package.name");
            StackTraceElement elements[] = e.getStackTrace();
            for (int i = 0, n = elements.length; i < n; i++) {
                logger.log(Level.WARNING, elements[i].getMethodName());
            }
        }
    }
}
```

Lancer une exception

- Exception lancée par l'instruction: **throw** *someThrowableObject*;
  - De nombreuses classes d'exceptions existantes




```
import java.util.*;
public class ListProducts {
    private int nb = 0 ;
    // …
    public Product pop() throws EmptyStackException  {
        if ( nb == 0 ) {
            throw new EmptyStackException();
        }
        return (Product) products.get( nb -1) ;
    }
}
```

- Possibilité de créer de nouvelles classes d'exceptions
  - si besoin d'une exception non représentée dans l'API Java, ou
  - pour différentier les erreurs issues de nos packages de celles issues des autres parties du code (et leur appliquer des traitements adaptés)

  ➢ Créer une classe héritant de la classe **Exception** (ou d'une classe fille)
  p.ex. les descendants de **RuntimeException** représentent plus particulièrement une mauvaise utilisation d'une API

Intérêt des exceptions

algorithme sans mécanisme gérant les exceptions

```
errorCodeType readFile {
   initialize errorCode = 0;

   open the file;
   if (theFileIsOpen) {
      determine the length of the file;
      if (gotTheFileLength) {
         allocate that much memory;
         if (gotEnoughMemory) {
            read the file into memory;
            if (readFailed) {
               errorCode = -1;
            }
         } else {
            errorCode = -2;
         }
      } else {
         errorCode = -3;
      }
      close the file;
      if (theFileDidntClose && errorCode == 0) {
         errorCode = -4;
      } else {
         errorCode = errorCode and -4;
      }
   } else {
      errorCode = -5;
   }
   return errorCode;
}
```

algorithme avec des exceptions

```
readFile {
   try {
      open the file;
      determine its size;
      allocate that much memory;
      read the file into memory;
      close the file;
   } catch (fileOpenFailed) {
      doSomething;
   } catch (sizeDeterminationFailed) {
      doSomething;
   } catch (memoryAllocationFailed) {
      doSomething;
   } catch (readFailed) {
      doSomething;
   } catch (fileCloseFailed) {
      doSomething;
   }
}
```

Exercice: Gestion des erreurs par des exceptions

- Intégrer dans le code de la classe bibliothèque une gestion des erreurs de saisies faites par les utilisateurs, p.ex.
  - gérer un emprunt de livre n'existant pas
  - gérer un client non enregistré

- Créer et utiliser une classe d'exception *PasEmpruntableException* déclenchée lorsqu'un document non empruntable tente d'être emprunté