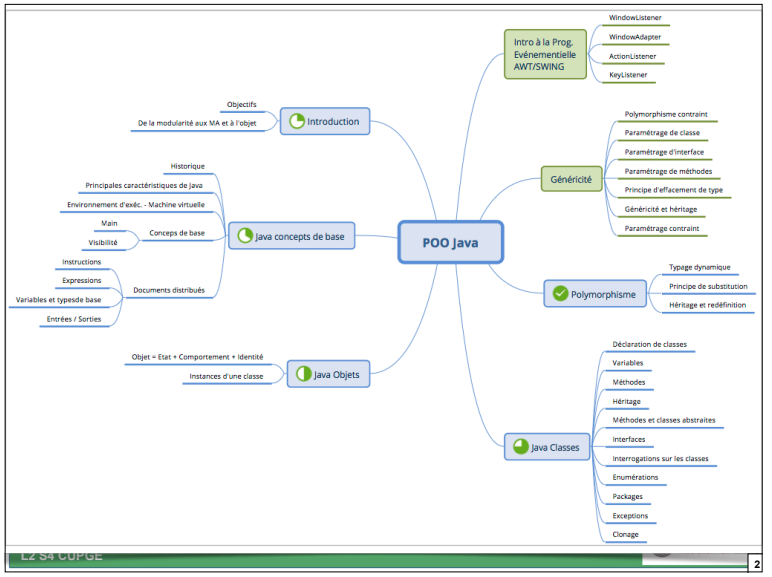
POO

Programmation Orientée Objet

UNIVERSITÉ TOULOUSE III PAUL SABATIER



Objectifs

- **Introduction à la Programmation Orientée Objet**
  - **Prérequis : programmation impérative et langage C**
    - Structure de contrôles
    - Sous-programmes
    - Types abstraits de données

  Difficulté à réutiliser le code

  - **Programmation Orientée Objet**
    - Collection d'objets simples
    - Construction d'objets complexes

  Réutilisabilité (JDK de JAVA)

But de la POO

- **Qualité du logiciel**
  - **Réutilisabilité, fiabilité, robustesse**
- **Qualité : difficile à mettre en oeuvre**
  - **30% du coût en développement**
  - **70% en maintenance !!**
- **Coût de la non qualité**
  - **Logiciel de commande et de contrôle de l'US air force**
  - **Système de réservation de United Airlines**
  - **Logiciels de gestion**
  - **Logiciels temps réels embarqués**
  - **Logiciels temps réels sols**

Modularité

- **Autour des données ou autour des traitements ?**
  - **Programme = Actions sur des Données**
    - Décomposition classique par les actions (fonctionnelle, sous-prog)
  - **Ou**
    - Décomposition par les données (modules, classes)
- **Exemple :**
  - **Trier des administrés d'une commune mais selon quel critère ?**
    - Nom
    - Age
    - Adresse
    - ?

Non à la décomposition fonctionnelle descendante

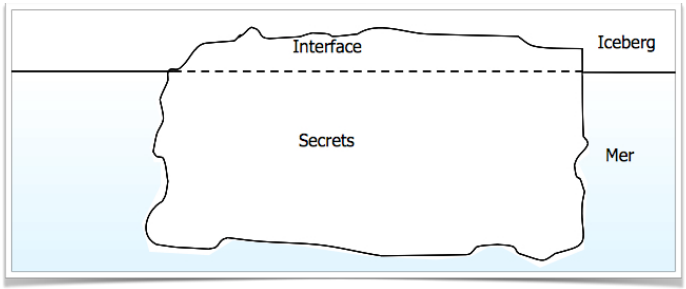
Oui à la composition ascendante

- **Partir des objets du système**
  - **Créer un module pour chaque type d'objets qui gère les données et les services**
  - **Développer des services compatibles**
  - **Penser réutilisabilité !**

Ne pas définir a priori ce que fait le système : définir SUR QUOI il agit !

Mise en oeuvre de la modularité

- **Interface publique**
- **Le reste doit rester caché !**



Un module = Unité de compilation

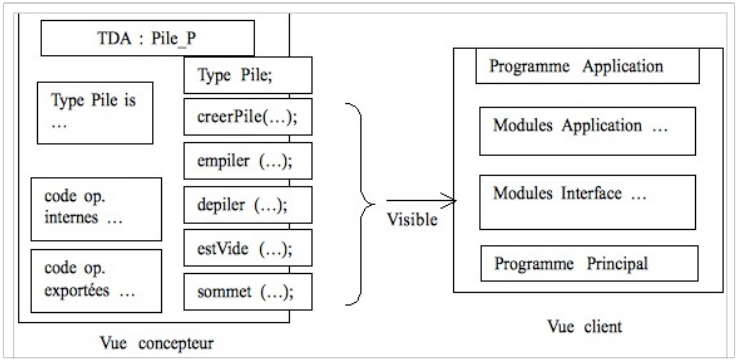
C : 2 fichiers	INTERFACE = nomModule.h (spécification des données et signature des opérations accessibles (sous-programmes)) CORPS = nomModule.c (code de **toutes** les opérations, déclarées dans le .h ou pas)
ADA : package	INTERFACE = spécification (types et données accessibles ou non et signatures des opérations accessibles) CORPS = body (code des opérations, déclarées dans la spécification ou pas)
Java : classe	CLASSE = attributs (données) et méthodes (sous-programmes) accessibles *public* ou non depuis l'extérieur.

Modularité -> Machines abstraites

- **Exemple de la Pile**
  - **Type de données abstraits (TDA)**
  - **Propriétés fondamentales**
    - Observation de piles réelles
  - **Spécifier le concept abstrait de Pile**
    - Indépendamment de l'implantation
    - Indépendamment des clients
  - **Choix d'implantation**
    - Langage cible
    - Structure de données
    - Opérations sur la Pile

TDA Pile

- ***Client* du TDA : application qui utilise le type Pile exporté par le TDA**

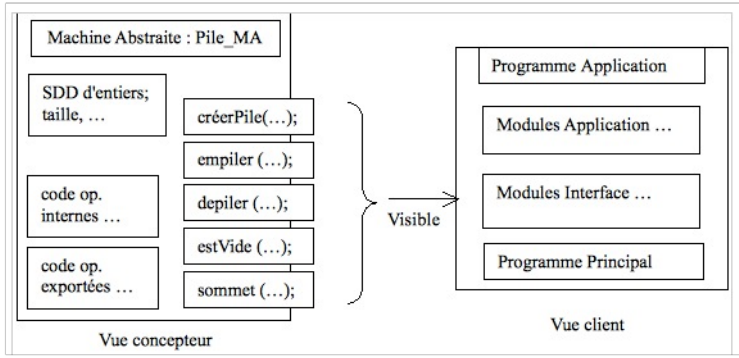


TDA Pile

- **V1 : Encapsulation**
  - **Pile.h**
    - déclaration du type structure Pile_s, qui n'est pas caché
    - signature des opérations
  - **Pile.c**
    - code des opérations
- **V2 : Encapsulation & Abstraction**
  - **Pile.h**
    - déclaration d'un type Pile, pointeur qui cache la structure réelle de la pile
    - signature des opérations
  - **Pile.c**
    - déclaration du type Pile_s décrivant la structure réelle de Pile
    - code des opérations

Machine abstraite Pile

- **Client de la MA : application qui utilise une pile gérée par la machine abstraite (machine à états abstraits)**



Machine abstraite Pile

- **Encapsulation & abstraction**
  - **PileMA.h**
    - n'exporte pas le type Pile
    - signature des opérations
  - **PileMA.c**
    - déclaration du type Pile réel
    - code des opérations
  - **Attention : 1 seule SDD Pile partagée par les clients**
- **De la programmation modulaire à la CPOO**
  - **Classes vues comme nouveaux *types***
    - Attributs et méthodes …

Java : Concepts de base

- **Historique**
  - **1993 : Protocole HTTP, premier navigateur internet (NCSA-Mosaic)**
  - **1995 : Premier environnement Java (compilateur, debugger, bibliothèque graphique et de communication, environnement d'exécution pour différentes plateformes)**
  - **1996-2002 : Evolution de l'environnement Java**
  - **2003 : Java 2 et généricité (polymorphisme contraint)**
  - **2014 : fonctions anonymes (lambda-expressions)**
- **IDE : Eclipse**
  - **eclipse.org -> Download + install SDK Java**
  - **https://docs.oracle.com/javase/8/docs/api/**
  - **http://help.eclipse.org/oxygen/index.jsp?nav=%2F1**

Principales caractéristiques

- **Langage à objets**
  - **Encapsulation et abstraction de données**
- **Langage de classes**
- **Syntaxe proche de C et C++**
- **Langage fortement typé**
  - **Gestion de la mémoire**
  - **Vérification à l'exécution**
- **Portabilité, indépendance du système**
  - **Compilation vers une machine virtuelle**
  - **Exécution sans recompilation**
- **Applications parallèles et distribuées**
- **SDK particulièrement riche**

Environnement d'exécution

- **JVM (Java Virtual Machine)**
  - **Instructions opérant sur registres et zones mémoire**
  - **Jeu d'instruction (byte-code)**
- **Programme Java compilé vers la JVM**
  - **Instructions interprétées par l'émulateur de la JVM de la machine support**
- **Exemple "hello world" ...**

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello world!");
        }
}
```

Visibilité

- Attention !! Pour les attributs visibilité en **RW**

Accès	Classe	Package	Sous-classe	Autres
public	Y	Y	Y	Y
protected	Y	Y	Y	N
package (defaut)	Y	Y	N	N
private	Y	N	N	N

Syntaxe de base

- Voir supports de cours et package java.lang
  - Instructions
  - Expressions
  - Variables
  - Types de base
  - Entrées / Sorties
- Fonction principale : main

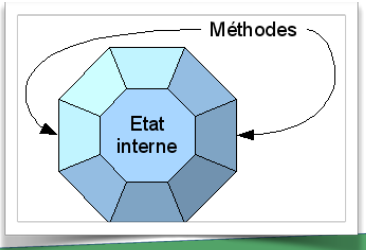
```
public  static   void     main ( String     arg [] ) {
   code ...
}
```

arguments sous forme de tableau de String
nom obligatoire de la procédure principale
procédure => pas de return
méthode de classe
visible par tous

Objets : définition

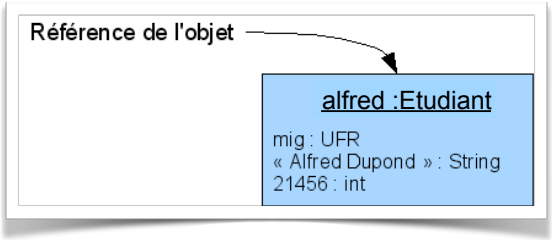
Objet = Etat + Comportement + Identité

- Etat interne
- Comportement = compétences
  - Agit sur l'état interne



Objet : Instance de classe

- Identité UNIQUE
  - Référence de la zone mémoire allouée lors de la création de l'objet
- Etat = valeurs des attributs



Classes : Déclaration

- **Classe simple**

```
public class MyClassName {
        // attributs
        // constructeurs
        // méthodes
}
```

- **Héritage**

```
public class MyClassName extends ClasseParente {
        ...
}
```

- **Mise en oeuvre d'interface**

```
public class MyClassName implements NomInterface1, NI2, ... {
        ...
}
```


Variable d'instance

- **Attribut dont la valeur est stockée dans chaque instance**

```
public class Velo {
    // var d'instance : "VTT", "VAE", etc.
    protected String bikeType;
    ...
}
```

```
protected MyClass myObject;
...
myObject.var = uneValeur;
myObject.otherObject.otherVar = otherValue;
          ↗          ↗          ↗
instance   instance   variable
```


Variable de classe

- **Attribut dont la valeur est stockée une fois dans la classe**
  - **Mot clé static**

```
public class FamilleDurand {
    // variable de classe
    protected static String nom = "Durand";
    // attributs
    protected String  prenom;    // variable d'instance
    protected int age;           // "
    ...
}
// client
FamilleDurand dad = new FamilleDurand();
System.out.println("nom : " + dad.nom);  // idem à
System.out.println("nom : " + FamilleDurand.nom);
```


Mot clé : final

- **Constante de classe**

```
protected static final float PI = 3.141592f;
```

  - **ou mieux, constante définie dans la classe Math de java.lang : double Math.PI**

- **Constance d'instance**

```
private final int MAX_SIZE = 4000000;
```

- **Convention d'écriture**
  - **MAJUSCULES_ET_SEPARATEUR '_'**

Constructeur, mot clé : new

- **Création d'une instance**

```
constructeur  = allocation + initialisation
```

- **Opérateur new**
- **Constructeur porte le nom de la classe**

```
Random choixHasard = new Random();
FamilleDurand fred = new FamilleDurand("Frédéric");
```

- **Constructeur par défaut**
  - **Constructeur sans paramètre**
  - **Initialise les attributs avec des valeurs par défaut**
    - instances à **null**, variables à des **valeurs par défaut**

Mot clé : this

- **this : référence l'instance courante**

```
public class FamilleDurand {
   protected static String nom = "Durand";
   protected String prenom;  // attribut
   public FamilleDurand (String prenom) { // constructeur
      this.prenom = prenom;   // obligatoire !!
      this.myMethod();            // méthode invoquée sur
                                     l'instance courante
   }
  public String getNom() {
     return FamilleDurand.nom;         // ou return this.nom;
   }
}
```


Méthode d'instance

- **Si ReturnType == void**

```
public void nomMethode( [type arg1, ... , type argn]) {
          ...   // pas de return
}
```

- **Si ReturnType != void**

```
public ReturnType  nomMethode( [type arg1, ... , type argn]) {
          ReturnType uneVariable  ;
          ...
          return uneVariable  ;    // obligatoirement un return
}
```

- **Si ReturnType est un tableau**

```
public int[] nomMethode( [type arg1, ... , type argn]) {
          int[] unTableau = new int[TAILLE]  ;
          ...
          return unTableau  ;// obligatoirement un return
}
```


Appel des méthodes d'instance

- **Méthode appliquée à l'instance**
- **Exemple classe String**

```
String str = "Mieux vaut tard que jamais.";
System.out.println ( "La chaîne est : " + str.toString() );
System.out.println ( "Longueur de cette chaîne : " + str.length( ) );
System.out.println ( "Le caractère en position 5 : " + str.charAt(5) );
System.out.println ("La sous-chaîne 11 à 18 : " +     str.substring(11, 18) );
System.out.println ("L'index du caractère d : " + str.indexOf( 'd' ) );
System.out.println ("L'index du début de la sous-chaîne \"tard\":" +
                    str.indexOf("tard") );
System.out.println ("La chaîne en majuscules : " + str.toUpperCase( ) );
//Si la méthode invoquée renvoie une instance de classe,
// on peut y appliquer à son tour une méthode
System.out.println ("La  longueur de la sous-chaîne 11 à 18 : " +
                     str.substring(11, 18).length() );
```

Appel des méthodes d'instance

- **Q : Résultat de l'exécution**

Méthodes de classe

- **Ne peuvent accéder qu'à des variables statiques !!!**

```
public static ReturnType nomMethode([args]) {
   ...
}
```

- **Exemple classe Math du package java.lang**

```
// sin est une méthode de classe et x une valeur réelle
double    f      = Math.sin ( x );
double    root   = Math.sqrt (453.0);
  int     max    = Math.max (x, y);
```


Ex. classe qui compte les instances

```
public class InstanceCounter {
  private static int instanceCount = 0;   // var de classe
  public InstanceCounter( ) { // constructeur
    InstanceCounter.incrementCount ( );  // incr. le compteur a la creation
  }
  protected void finalize ( ) {   // définie dans Object -> decremente
    instanceCount = instanceCount - 1;
  }
  public static int instanceCount ( ) {
    // methode d'acces a la variable de classe
    return instanceCount;
  }
  private static void incrementCount ( ) {
    instanceCount = instanceCount + 1;
  }
}
```


Ex. classe qui compte les instances

```
public class Main {
  public static void main(String args[]) {
    System.out.println("nbr inst = " + InstanceCounter.instanceCount());
    InstanceCounter ic = new InstanceCounter();
    System.out.println("nbr inst = " + InstanceCounter.instanceCount());
    InstanceCounter ic1 = new InstanceCounter();
    System.out.println("nbr inst = " + InstanceCounter.instanceCount());
    ic1 = new InstanceCounter();
    Runtime.getRuntime().gc();  // déclenchement du garbage collector
    Runtime.getRuntime().runFinalization();  // et de finalize
    System.out.println("nbr inst = " + InstanceCounter.instanceCount());
  }
}
```

Mode de transmission des paramètres

Types de base ➥ par valeur

Instances ➥ par référence

Surcharge

- **Même nom de méthode, paramètres différents**
  - **Attention : type de retour non testé par le compilateur**
- **Dans la même classe**
- **Dans une classe héritière (à venir)**

Surcharge : exemple

```
import java.awt.Point;  // import de Point depuis le package awt
public class MyRect {
  protected Point  p1, p2;
  public MyRect() {
    p1 = new Point(0,0);
    p2 = new Point(0,0);
  }
  public void buildRect ( int x1, int y1, int x2, int y2 ) {
    this.p1.setLocation ( x1, y1);
    this.p2.setLocation ( x2, y2);
  }
  public void buildRect ( Point topLeft, Point bottomRight ) {
    p1 = topLeft;   // Attention : affectation de références
    p2 = bottomRight;
  }
                                                    ...
```


Surcharge : exemple

```
  public String toString() {
    return "["+p1+", "+p2+"]";
  }
  public static void main (String args [] ) {
    MyRect  rect = new MyRect ( );  // constructeur par défaut
    pTL = new Point (10,10);
    pBR = new Point (20, 20);

    rect.buildRect (20, 30, 40, 50 );
    System.out.println(rect);
    rect.buildRect (pTL, pBR);
    System.out.println(rect);
  }
}
```


Surcharge des constructeurs

```
public class Personne {
// surcharges de constructeurs
  protected String name;
  protected int age;

  public Personne (String s, int a ) {
    name = s;
    age = a;
  }

  public Personne (String s ) {
    name = s;
    // age est initialisé par défaut a 0
  }

// necessaire ici pour la creation de
// p3 avec valeurs par defaut
  public Personne() {
      name = "";
  }

  public String toString () {
    return "["+name+", "+age+" ans]";
  }

  public static void main (
                          String args [] ) {
    Personne p1, p2, p3;
    p1 = new Personne ("Laura", 20 );
    System.out.println(p1);
    p2 = new Personne ("Laurie");
    System.out.println(p2);
    p3 = new Personne ( );
    System.out.println(p3);
  }
}
```

Q : Résultat ?

Constructeur par copie

- Pour dupliquer une instance
  - constructeur avec une instance en entrée
  - retourne une copie de cette instance
- Exercice
  - Ecrire le constructeur par copie de la classe MyRect

Exercice : MotorCycle

- Ecrire le code de la classe MotorCycle
  - Au moins un constructeur
  - Méthode qui démarre le moteur, uniquement s'il n'est pas déjà démarré
  - Méthode toString qui retourne l'état de l'instance
    - public String toString(); // définie dans la classe Object
  - Méthode equals qui teste l'égalité de deux instance
    - public boolean equals(Object obj) { // définie dans la classe Object
      - MotorCycle mc = (MotorCycle) obj;
      - // …. code qui teste l'égalité entre le **this** et **mc**
    - }
  - main pour tester

TD : Classe MySet, Ensemble (de float)

- Cliente de la classe ArrayList (java.util)
- Méthodes de la classe :
  - Un constructeur sans paramètre, qui crée un ensemble vide
  - add : ajoute la valeur passée en paramètre (VPP) si non présente
  - size : renvoie le nombre de valeurs de l'ensemble
  - isEmpty : renvoie true si l'ensemble est vide, false sinon
  - remove : supprime la VPP de l'ensemble
  - contains : renvoie true si la VPP est dans l'ensemble, false sinon
  - toString : revoie une chaîne de caractère (état de l'ensemble)
  - clear : supprime toutes les valeurs de l'ensemble
  - random : renvoie une des valeurs de l'ensemble (au hasard)
  - equals : teste si l'ensemble passé en paramètre contient les mêmes valeurs que le this

TDA Ensemble (de float)

- Obligatoire : consulter la javadoc de la classe ArrayList !
- Utilise : boolean, float
- Opérations :
  - cons : -> Ensemble
  - add : Ensemble x float -> Ensemble
  - remove : Ensemble x float -> Ensemble
  - isEmpty : Ensemble -> boolean
  - size : Ensemble -> int
  - contains : Ensemble x float -> boolean
  - clear : Ensemble -> Ensemble
  - random : Ensemble -> float
- Pré-conditions : s : Ensemble, f : float
  - random() = f => ! isEmpty(s)
- Axiomes : s : Ensemble; e, f : float
  - (P1) contains(add(s,e), e) = true
  - (P2) remove(add(s,e)) = s
  - (P3) isEmpty(cons(s)) = vrai
  - (P3) isEmpty(add(s,e)) = faux
  - (P4) add(cons(s),e) = add(remove(add(cons(s),f)), e)
  - (P5) size(cons(s)) = 0
  - (P6) size(add(s, e)) = size(s) + 1
  - (P7) size(remove(s, e)) = size(s) - 1

Héritage

- **Hériter :**
  - **Pour une classe fille = disposer des attributs et méthodes de la classe parente**
- **ATTENTION : En Java Héritage SIMPLE !!!**
  - **Une classe hérite au maximum *d'une seule* classe**
  - **Plus : héritage implicite de Object**
    ----------->>> *ne pas* écrire *explicitement*
    ----------->>> class X ~~extends Object~~
    - **Object** : classe parente implicite de **toute** classe
    - Voir l'API :
      - **public** String toString() { // définition par défaut }
      - **public** boolean equals(Object obj) { // définition par défaut}

Héritage

- **Déclaration mot clé extends**

```
class MyClassName extends ClasseParente {
        ...;  // code
}
```

- **Mot clé super**
  - **Référence la classe parente**

```
class Auto extends Vehicule {
  ...
    public void faireLePlein () {  // méthode surchargée
        compteurJournalier = 0; // déclaré dans Véhicule donc visible
        super.faireLePlein ();    // fait appel à la méthode faireLePlein
                                  // de la classe Véhicule
    }
```


Héritage

- **Mot clé super**
  - **Cas du constructeur**
    - Appel ***OBLIGATOIRE*** du constructeur du parent en première ligne :

```
super ([paramètres]);
```

```
import java.awt.Point;
public class NamedPoint extends Point {
    protected String name;
    public NamedPoint (int x, int y, String n) {
        super (x, y);     // appelle le const. de la classe Point
        this.name = n;
    }
}
```


Redéfinition

- **Si B hérite de A alors**
  - **B hérite de tous les attributs et méthodes visibles**
    - public
    - protected
    - package (défaut) **SI** dans le même package
  - **B hérite de toutes les méthodes *pré*définies dans Object**
    - toString : retourne par défaut l'@ hexa de l'instance en mémoire
    - equals : compare les @ en mémoire du **this** et du paramètre
- ***Redéfinir* toutes les méthodes qui ne conviennent pas !!**
- **——>>>>> ATTENTION : surcharge != redéfinition**
  - **surcharge : signature *différente***
  - **redéfinition : *même* signature**

Redéfinition

```
public class A {
    float donneeDeA ;
    ...
    public String toString() {
        return "A : "+ donneeDeA ;
    }
}
public class B extends A {
    int donneeDeB ;
    ...
    public String toString() {  // redéfinition car ne convient pas
        return super.toString() + ", B : " + donneeDeB ;
    }
}
```


Méthodes et classes abstraites

- Mot clé : *abstract*
- Si *au moins 1* méthode abstraite alors classe abstraite
- Une classe abstraite *ne peut pas* être instanciée

```
public abstract class Figure {
  // attributs
  ...
  // Constructeurs
  ...
  // Méthodes
  public abstract void draw() ;
  ...
}
```

```
public class Triangle extends Figure {
  // code de la méthode abstract
  public void draw () {
     // code de tracé d'un triangle
          ...
  }
  ...
}
```


Interfaces

- Ne contient *que* des signatures de méthodes
  - PAS d'attributs
  - PAS de code
- Décrit un comportement
- Permet de pallier les inconvénients de l'héritage simple

```
public interface IPileEntiers {
    public void add(int val);  // ajoute la valeur sur le sommet
    public int get() ;         // renvoie la valeur au sommet
    public void remove() ;     // supprime la valeur au sommet
    public int size() ;        // renvoie la taille de la pile
    public int capacity() ;    // renvoie la taille max de la pile
}
```


Interfaces

- Une classe peut mettre en oeuvre plusieurs interfaces

Method Summary

int	**compareTo**(T o) Compares this object with the specified object for order.

```
public class CompPile implements IPileEntiers,
                                 Comparable<CompPile> {
   // attributs
   ...
   // constructeur
   ...
   // méthodes de IPileEntiers
   ...
   // méthode de Comparable
   int compareTo(CompPile cp) { ... }
}
```

Interfaces et Héritage

- Une classe peut hériter UNE fois
- Une classe peut mettre en œuvre plusieurs interfaces

```
public class ColoredRect extends Rectangle
                    implements Comparable<ColoredRect>  {
    // attributs de Rectangle +
    Color rectColor;
    // constructeur
    ...
    // méthode de Comparable
    int compareTo(ColoredRect cr) { ... }
}
```


Exercice Mise en œuvre de IPileEntiers

- Ajouter à la classe deux constructeurs et la mise en œuvre des méthodes héritées de la classe Object :
  - Un constructeur qui crée une pile vide de taille tailleMax
  - Un constructeur par copie
  - toString : affiche le contenu de la pile
  - equals : renvoie true si la pile passée en paramètre contient les mêmes valeurs dans le même ordre, false sinon

Type Abstrait : Pile (de int)
Utilise : boolean, int
Opérations :
- cons : -> Pile
- copie : Pile -> Pile
- add : Pile x int -> Pile
- remove : Pile -> Pile
- get : Pile -> int
- size : Pile -> int

Pré-conditions : s : Pile, f : int
- get() = f => size(s) > 0

Axiomes : s : Pile; e, f : float
(P1) size(add(s,e), e) > 0
(P2) remove(add(s,e)) = s
(P3) isEmpty(cons(s)) = vrai
(P3) isEmpty(add(s,e)) = faux
(P4) add(cons(s),e) =
  add(remove(add(cons(s),f)), e)
(P5) size(cons(s)) = 0
(P6) size(add(s, e)) = size(s) + 1
(P7) size(remove(s, e)) = size(s) - 1

Interrogation sur les classes

- Class uneClasse = Class.forName("className")
  - retourne la classe correspondant à l'argument
- Class parente = instance.getSuperClass();
  - retourne la classe parente de l'instance
- String name = instance.getClass().toString();
  - retourne le nom de la classe de l'instance
- instanceof teste la classe d'une instance

```
("foo" instanceof String)  -- true
Point pt = new Point (10, 10);
(pt instanceof String)     -- false
```


Enumérations

- Ex. Season.java
- Testé par :

```
public enum Season {
    SPRING, SUMMER, AUTOMN, WINTER;
}
```

```
public [static] void testSeason (Season oneSeason) {
    switch (oneSeason) {
    case SPRING:
        System.out.println("Les arbres sont en fleurs !!!");
    break;
    case SUMMER:
        System.out.println("Il fait chaud !!!");
    break;
    case AUTOMN:
        System.out.println("Les feuilles tombent...");
    break;
    case WINTER:
        System.out.println("Il fait froid !!!");
    break;
    }
}
```

Enumérations

- Héritent implicitement d'une classe java.lang.Enum
- Valeurs ordonnées, la première au rang 0
- Méthodes les plus utilisées
  - **val.ordinal()**
    - retourne la position de val dans l'énumération
  - **String toString()**
    - retourne la ch. de carac. correspondant à la valeur courante
  - **static values()**
    - retourne un tableau des valeurs ordonnées de l'enum
  - **valueOf(String name)**
    - retourne une valeur correspondant à la ch. de carac
  - **values()**
    - retourne un tableau des valeurs ordonnées de l'enum

Enumérations

- Appel de testSeason
  - Cas où testSeason de la classe Dummy est static

```
Dummy.testSeason(Season.SUMMER);

System.out.println (Season.valueOf("spring"));

Season se[] = Season.values();
for (Season ses : se) {
    System.out.println(ses + ", ") ;
}
```

  - Résultat ?

Enumération plus complexe

```
import java.awt.Color;
public enum TypeBateau {
 // valeurs
 CUIRASSE(4, "cuirasse", Color.BLUE),
 CROISEUR(3, "croiseur", Color.CYAN),
 TORPILLEUR(2, "torpilleur", Color.RED),
 SOUS_MARIN(1, "sous-marin", Color.GREEN);
 // attributs d'une valeur
 public final int taille;
 public final String name;
 public final Color c;

 // constructeur appelé par l'enum
 TypeBateau(int taille, String name, Color c) {
   this.taille = taille;
   this.name = name;
   this.c = c;
 }

 // accesseurs publics

 public int getTaille(){
    return this.taille;
 }

 public String getName(){
   return this.name;
 }

 public Color getColor() {
   return this.c;
 }
}
```


Enumération plus complexe

- Constructeur de Bateau

```
public class Bateau {
   protected TypeBateau tn;
   protected boolean horizontal;
   public Bateau(TypeBateau typeBat, boolean hor) {
      tn = typeBat;
      horizontal = hor;
      int taille = typeBat.getTaille();
      …
   }
   …
}
```

Enumération plus complexe

- Méthode d'une classe Joueur qui utilise TypeBateau

```
public void addBateau(TypeBateau tn, Point p, boolean hor) {
   Bateau unBateau = new Bateau(tn, hor);
   ....
}
```

- Création d'un TypeBateau

```
TypeBateau unTypeBateau = TypeBateau.CUIRASSE;
```

- Appel de la méthode

```
joueur1.addBateau(unTypeBateau, unPoint, true);
```


Packages

- Classes du SDK de Java organisées en packages

Packages

- Classes de java.lang importées implicitement
- Toute classe d'un autre package : import explicite

```
import java.awt.Button;
public class UseAwt {
   protected Button b;      // classe de java.awt
   ...
}
```

- Packages :
  - Groupement thématique de classes et interfaces
  - Contrôle de l'accès
  - Evite les conflits de noms

Packages

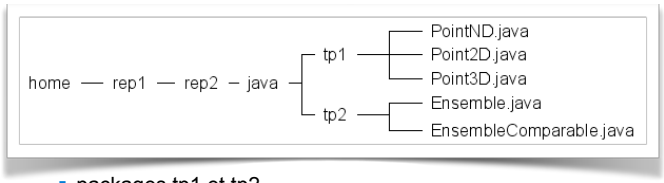
- Création de ses propres packages
  - Organisation des classes en hiérarchie de répertoires
  - Export explicite d'une classe
  - Ex : la classe Point2D du package tp1

```
package tp1;
public class Point2D {
   // attributs de la classe Point2D
   ...
   // etc.
}
```

  - Toutes les classes du package tp1 doivent être dans le même répertoire de nom tp1

Packages

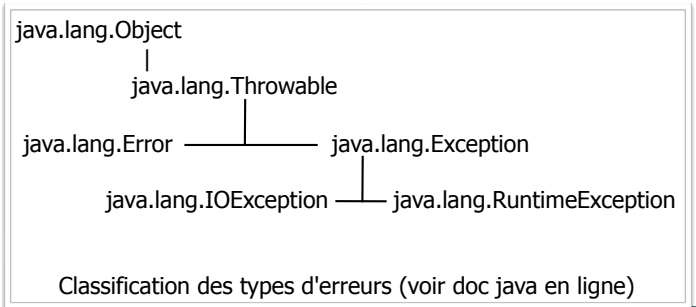
- **CLASSPATH**
  - **Variable d'environnement qui indique le chemin d'accès aux répertoires Java**
  - **Exemple d'organisation**



  - packages tp1 et tp2
  - chemin d'accès commun : /home/rep1/rep2/java
  - setenv CLASSPATH .:/home/rep1/rep2/java
- **Géré automatiquement par eclipse ….**

Exceptions

- **Programmation par contrat**
- **Gestion des erreurs**
- **Hiérarchie de classes, dérivées de Throwable**



Classification des types d'erreurs (voir doc java en ligne)

Exceptions

- **Constructeurs**

Constructor Summary	
Throwable ()	Constructs a new throwable with null as its detail message.
Throwable (String message)	Constructs a new throwable with the specified detail message.
Throwable (String message, Throwable cause)	Constructs a new throwable with the specified detail message and cause.
Throwable (Throwable cause)	Constructs a new throwable with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause).

Créer une classe Exception

```
public class MyException extends Throwable (
                        // ou toute autre classe dérivée d'Exception
    public MyException( ) {
        ...
    }
    public MyException(Type1 a1, Type2 a2, ...) {
        ...
    }
}
...
// dans du code connaissant MyException
    ...
    if (cond)  {
        throw new MyException( );
    }
   ...
```

Récupérer une Exception

- Bloc try-catch

```
try {
    // code à exécuter susceptible de lever une exception
    // (voir API pour savoir quelles sont les
    // méthodes pouvant lever une exception
    // et la classe de cette exception)
} catch (MyException me) {
    // ici traitement de l'exception me
} catch (IOException ioe) {
    // pour tester un autre type d'exception
    // traitement de l'exception ioe
}
```


Lever (déclencher) une Exception

- 3 façons de lever une Exception
  - volontairement, throw new xxxException([arg])
  - appel d'une méthode qui lève une Exception
  - erreur interne Java (Error)
- 2 grandes catégories d'Exceptions
  - RuntimeException
    - ne nécessite **pas** de clause **throws**
    - n'oblige pas le client à encadrer l'appel par un bloc **try-catch**
    - doit amener à la correction de la cause de don déclenchement
  - IOException
    - clause **throws** obligatoire
    - le client **doit** encadrer l'appel par un bloc **try-catch**

Clauses throws

- Renvoi explicite d'une Exception

```
public String readLine() throws IOException { … }
```

- Gestion de l'Exception obligatoire

```
// par un bloc try-catch
public void methodeA() {
    try {
        return myReader.readLine();
    catch (IOException ioe) {
        // traitement de l'exception
    }
}
// ou par une transmission à l'appelant avec la clause throws
public void methodeB() throws IOException {
    String dummy = myReader.readLine();
}
```


Clause finally

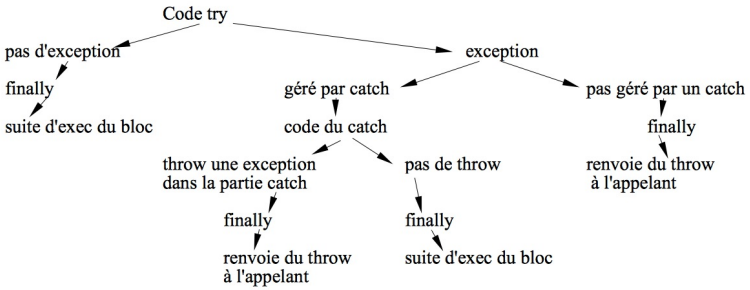
- Exemple :

```
Graphics g = image.getGraphics ( );
try {
    // code sur g
} catch (IOException e) {
    done = true; // par exemple
} finally {
    g.dispose ( );     // libération du graphic context
}
```

- Clause finally **toujours** exécutée

Clause finally

- Toujours exécutée :



Clonage

- Interface Cloneable
- Redéfinition de la méthode de Object

```
protected Object clone() throws CloneNotSupportedException;
```

- MAIS Clonage à un seul niveau …
  - Attention aux alias après le premier niveau.
  - Idem avec le constructeur par copie !

Clonage

- Clonage récursif ...

```
public class Test implements Cloneable {
   // toute donnée de base sera clonée par l'appel de super.clone()
   int donneeBase;
   // toute donnée instance de classe est clonée en appelant
   // SA méthode clone qui doit donc être implantée
   // ArrayList possède de base une méthode clone
   protected ArrayList<Data> donnees = new ArrayList<Data>();
   // Data doit être CLoneable !!
   ......
   protected Object clone() throws CloneNotSupportedException {
      Test copie = (Test)super.clone();
      copie.donnees = (ArrayList)donnees.clone();
      return copie;
   }
}
```


Exemple : AgentSecret

```
public class AgentSecret implements Cloneable {
   public Date dateNaissance;
   public String nom;
   public String prenom;
   public String service;
   public AgentSecret(Date date, String nom, String prenom,
         String service) { … }
   public void setService(String service) { … }
   public void setIdentite(String nom, String prenom) { …  }
   public void setDate(int day, int month, int year) { … }
   protected Object clone()  throws CloneNotSupportedException {
      AgentSecret copie = (AgentSecret)super.clone();  // Clone 1er niveau
      copie.dateNaissance = this.dateNaissance.clone(); // Date Cloneable
      return copie;
   }
   public String toString() { … }
}
```