Programmation Objet Avancée

- Assertions
- Structures de données
- Généricité
- Classe Abstraite
- Interface
- Classe interne
- Clonage

-1-

Assertions

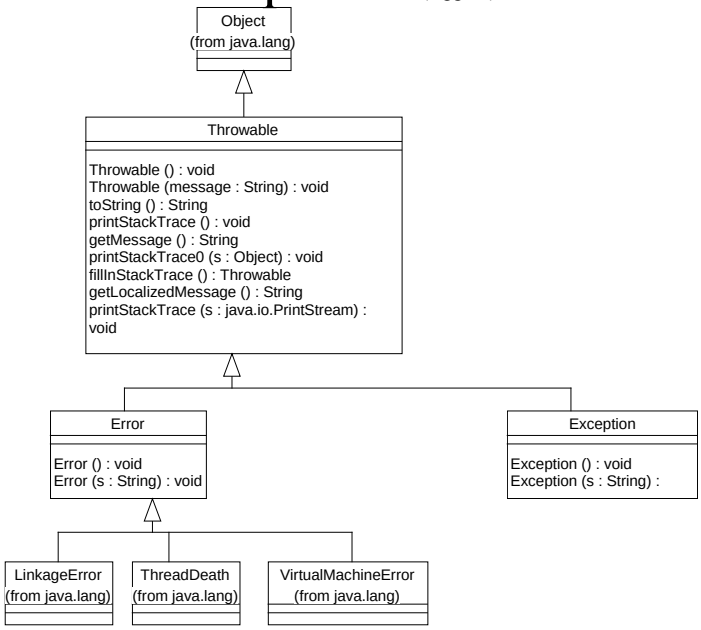
- Lors du dernier cours on a vu
  - Les instructions
  - Les exceptions

- Les assertions sont des instructions susceptibles de lever des exceptions !

-2-

Exceptions (rappels)

java.lang



l'instruction assert

- Depuis Java 1.4
- Destiné à capturer facilement des erreurs ssémantiques du programme.
- Disposition qui peut-être invalidée quand on le souhaite

Assert : principe

```
void credit(long amount)
{

   ...code ....

   assert   balance == amount + oldBalance
      &&
            transactions == oldTransactions+1
      &&
            balance < maxBalance ;

}
```

-5-

Syntaxe

- Deux formes pour assert :
  1. assert *booleanExpression*;
     - Teste l'expression booléenne
     - Ne fait rien si l'expression est évaluée à true
     - Si évaluée à false, cette instruction lève une AssertionError
  2. assert *booleanExpression* : *expression*;
     - Comme précédant
     - En plus si false, la seconde expression est évaluée comme message de l'exception AssertionError
     - Cette seconde expression peut être de n'importe quel type sauf void

-6-

Exemples

- assert 0<=i && i<=size : “out of range:”+i;
- assert p!=null : “illegal reference”;
- assert nextPerson()!=null : “not enough”;
- assert isSorted(this) : “unsorted” ;

-7-

Les Messages peuvent être des objets

- assert x>0 : “bad x” + x ;
- assert x>0 : “nasty mess at “ + new Date() ;
- assert x>0 : this ;
- assert age < this.parent.age :
  “bad age reln:” +parent ;

-8-

Attention à l'écriture d'assertions

- Une AssertionError est une Error, pas une Exception
  - On n'est pas obligé de mettre un bloc try
  - Ni donc de bloc catch
  - Donc pas de travail suplémentaire induit
- La seconde expression n'est pas obligatoire
  - On récupère de toute façon la pile des appels et le numéro de la ligne dans le code source.
  - La seconde expression doit donner une information utile.

-9-

Exemples

```
if (x < 0) {
    . . .
}
else if (x == 0) {
    . . .
}
else {
    assert x > 0;
    . . .
}
```

```
switch(suit) {
   case Suit.CLUBS:
      …
      break;
   case Suit.DIAMONDS:
      …
      break;
   case Suit.HEARTS:
      …
      break;
   case Suit.SPADES:
      …
     break;
   default:
     assert false: suit;
}
```

-10-

Assertions *vs* Exceptions

- Quand utiliser une assertion plutot qu'une exception?
  - Les deux lèvent des problèmes, mais...
  - Le but et l'usage sont très *differents!*

- Une Exception signale à l'utilisateur un problème.
- Une assertion prévient le développeur d'un bogue.

- On crée des Exceptions pour traiter des problèmes dont on sait qu'ils peuvent survenir.
- On écrit des assertions pour affirmer des choses que vraies dans le programme à cet endroit...

-11-

Quand lever des Exceptions ?

- Quand vous:
  - Testez si les paramètres d'une méthode *public* ou d'un constructeur *public* sont ok.
    - Si c'est public …
  - Dans les entrées/sorties
    - Vous ne pouvez pas être sûr du fonctionnement.

- Vous devez,
  - *Penser chaque classe comme si vous n'écriviez que celle-ci.* Donc tout ce qui n'est pas du ressort de cette classe relève d'une Exception

-12-

Quand utiliser les assertions ?

- souvent!
- Elles sont faciles à écrire
  - Example: `assert age >= 0;`
  - Les Assertions fournissent une *documentation*, et pas seulement du test d'erreurs
  - Les Assertions sont une documentation qui peut être testée à l'exécution !

-13-

Invariants

- Un invariant est une expression qui doit toujours être vraie
- Un invariant interne est un fait dont vous pensez qu'il est vrai en un certain point du programme
  - `assert x > 0;`
- Un invariant de contrôle de la séquence de code : par exemple une assertion qui stipule qu'un point ne doit pas être atteint
  - `assert false: suit;`
- Un invariant de classe est une assertion qui doit être satisfaite par tout objet de la classe
  - Example: `assert person.age >= 0 && person.age < 150;`

-14-

Compilation avec Assertions

Pour compiler avec assertions ajouter le flag suivant

javac **-source 1.4** MyClass.java

-15-

Execution et assertions

- Les Assertions peuvent être invalidées
- Par défaut les assertions ne sont pas validées
- Pour les valider utiliser le flag
  - -enableassertions
  - (ou -ea) dans la ligne de commande java
- On peut valider ou invalider fichier par fichier
- java -ea:com.fruitbat... -da:com.wombat.Brickbat BatTutor

-16-

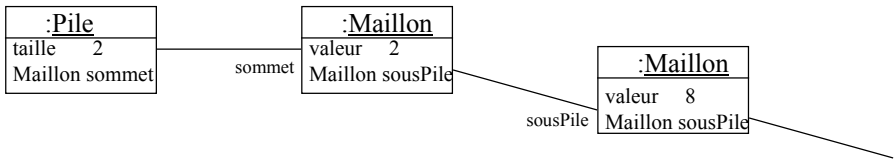
Structures de Données Récursives

- Application à un exemple de pile
- Structure de donnée générique
- Généricité, spécialisation et héritage

-17-

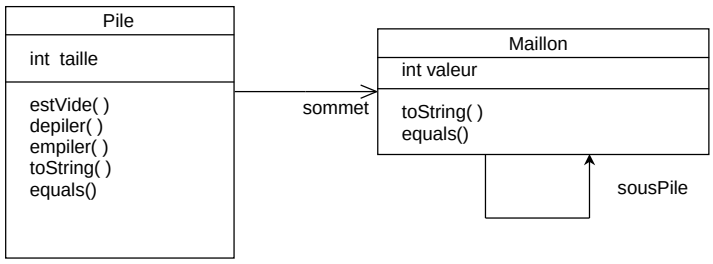
Pile de taille « illimitée »

modélisation objet d'une pile de taille «illimitée»



-18-

Modélisation UML



Code pour : Maillon

```
class Maillon{
    int valeur;
    Maillon sousPile;

    Maillon(int valeur,Maillon sousPile){
      this.valeur=valeur;
      this.sousPile=sousPile;
    }

    public boolean equals(Object that){
      return   this.valeur == ((Maillon)that).valeur
      &&    (this.sousPile==((Maillon)that).sousPile  || //==null
             this.sousPile.equals(((Maillon)that).sousPile)) ;
    }

    public String toString(){
      return valeur + ((sousPile!=null)? ", "+ sousPile : "");
    }                                          //toString()
}
```

Code pour : Pile

```
class Pile{
    protected int taille =0;
    protected Maillon sommet;

    public void empiler(int v){
      sommet=new Maillon(v,sommet);
      taille++;
    }
    public int depiler(){
      int v=sommet.valeur;
      sommet=sommet.sousPile;
      taille--;
      return v;
    }
    public boolean estVide(){
      return taille==0;
    }
    public String toString(){
      return "[ "+ sommet.toString() +" ]";
    }
    public String equals(Object that){
      return this.taille==((Pile)that).taille &&
             this.sommet.equals(((Pile)that).sommet);
    }
}
```

-21-

Généricité et la classe Object

On veut maintenant généraliser notre pile afin qu'elle puisse servir de pile d'objets et non plus seulement de pile d' « int ».

-22-

Maillon d'Object

```
class Maillon{
   Object valeur;
   Maillon sousPile;

   Maillon(Object valeur){
        this.valeur=valeur;
   }
   public boolean equals(Object that){
        return   this.valeur.equals((Maillon)that).valeur)
              && this.sousPile.equals((Maillon)that).sousPile);
   }
   public String toString(){
        return valeur + ((sousPile!=null)? ", "+ sousPile : "");
              //valeur.toString()
  }
}
```

-23-

Pile d'Object

```
class Pile{
   int taille =0;
   Maillon sommet;

   public void empiler(Object v){
         Maillon nouveauSommet=new Maillon(v);
         nouveauSommet.sousPile=sommet;
         sommet=nouveauSommet;
         taille++;
   }
   public Object depiler(){  //throws NullPointerException
         Object v=sommet.valeur;
         sommet=sommet.sousPile;
         taille--;
         return v;
   }
   public boolean estVide(){
         return taille==0;
   }
   public String toString(){
         …
   }
   public String equals(Object that){
         …
   }
}
```

-24-

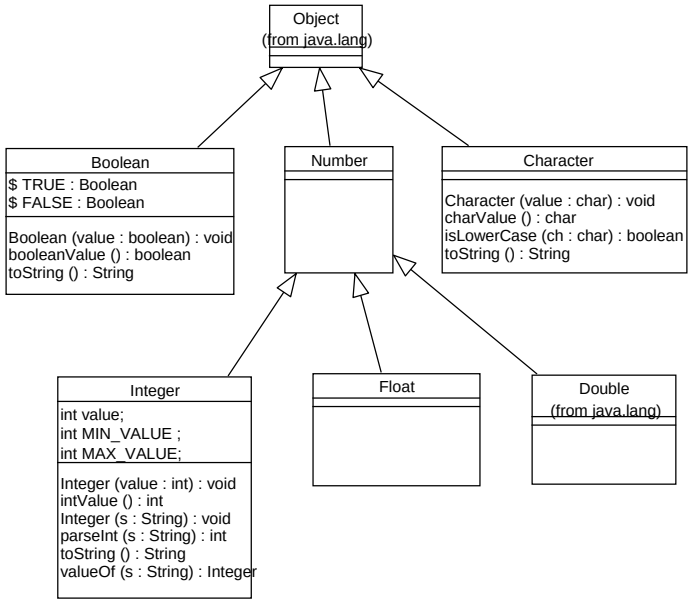
Empiler les chaines

```
public class PileChaine {

   public static void main(String[] args){
        Pile pile= new Pile();
        int i=0;
        while(i<args.length){
            pile.empiler(args[i]);              // on empile un objet String
           System.out.println(i+": "+ args[i]);  //  polymorphisme
           i++;
        }
        i=0;
        while(i<args.length){
           System.out.println(i+": "+ pile.depiler());  //polymorphisme
           i++;
        }
   }
}
```

-25-

Wrappers des types de base



-26-

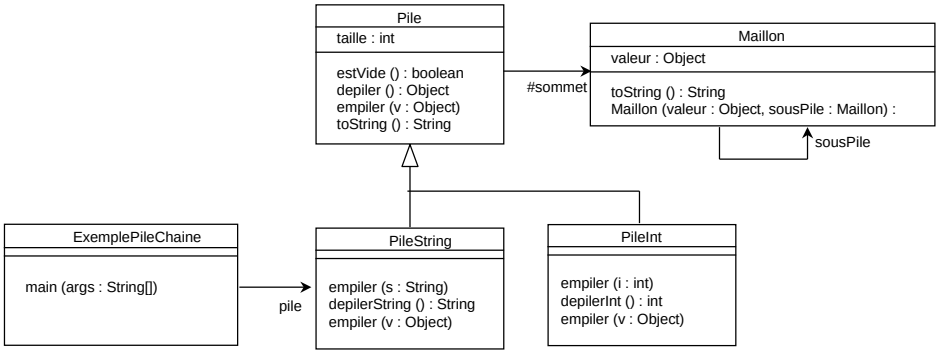
Sous-classe:
généricité et polymorphisme

```
public class PileChaine {

   public static void main(String[] args){
         Pile pile= new Pile();
         int i=0;
         while(i<args.length){
            pile.empiler(args[i]);                    // on empile un objet String
            System.out.println(i+": "+ args[i]);
            i++;
         }
         pile.empiler(new Integer(3));          // on empile un objet Integer
         pile.empiler(new Float(3.0));
         i=0;
         while(!pile.estVide()){
            System.out.println(i+": "+ pile.depiler().toString());  //polymorphisme
            i++;
         }
   }
}
```

-27-

Pile Générique et héritage



-28-

Sous Classe, héritage et surcharge

```
class PileString extends Pile{

   public void empiler(String s){
         super.empiler(s);
   }

   public void empiler(Object v){
         if(v instanceof String) super.empiler(v);
         else System.out.println("c'est une pile de String et non de "+v.getClass());
   }

   public Object  depiler(){    // deux méthodes ne peuvent différer par le seul type de retour
         return (String)super.depiler();
   }
}
```

-29-

Autre exemple

```
class PileInt extends Pile{

   public void empiler(int i){     // surcharge
         super.empiler(new Integer(i));
   }

   public void empiler(Object v){
         if(v instanceof Integer) super.empiler(v);
         else System.out.println("c'est une pile d'Integer et non de "+v.getClass());
   }

   public int depilerInt(){         // deux méthodes ne peuvent différer par le seul type de retour
         Integer i=(Integer)super.depiler();
         return i.intValue();
   }
}
```

-30-

Pile d'Objets de même type

```
class Pile{
   int taille =0;
   Maillon sommet;
   Class class;

  public Pile(Class class){  this.class=class;   }
  public void empiler(Object v){
        if(v.getClass()==class){
                Maillon nouveauSommet=new Maillon(v);
                nouveauSommet.sousPile=sommet;
                sommet=nouveauSommet;
                taille++;
        else throw new IllegalArgumentException();
   }
   public Object depiler(){
        …
   }
   public boolean estVide(){
        return taille==0;
   }
   public String toString(){
        …
   }
   public String equals(Object that){
        …
  }
}
```


Java 1.5 et la généricité

```
class Pile <E> {
   int taille =0;
   Maillon<E> sommet;

  public void empiler(E v){
                Maillon <E> nouveauSommet=new Maillon(v);
                nouveauSommet.sousPile=sommet;
                sommet=nouveauSommet;
                taille++;
   }
   public E depiler(){
        …
   }
   public boolean estVide(){
        return taille==0;
   }
   public String toString(){
        …
   }
   public boolean equals(Object that){
        …
  }
}
```

Classe Abstraite

```
public class PileAbstraiteTest {
   public static void main(String[] args){
            Pile pile= new PileString();          // la classe est abstraite mais l'instance non
           ...
    }
}
class Maillon{...
}

abstract class Pile{
   int taille =0;                                  // variable
   abstract public void empiler(Object v);         //signature
   abstract public Object depiler();
   public boolean estVide(){                       //implémentation
           return taille==0;
   }
}

class PileRécursive extends Pile{
   Maillon sommet;
   public void empiler(Object v){...    }
   public Object depiler(){...    }
   public String toString(){...    }
}

class PileString extends PileRécursive {
   ….
}
```

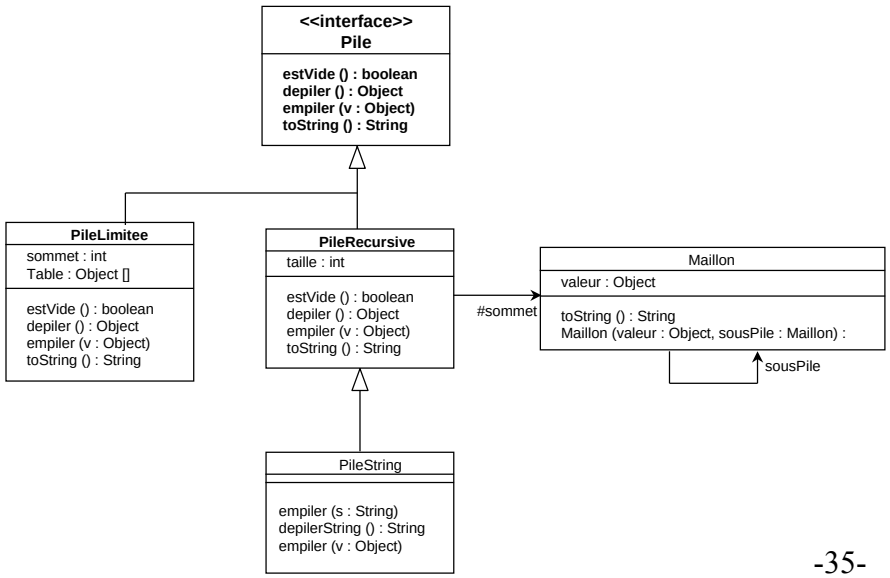
-33-

Interface

```
interface Pile{
   void empiler(Object v);
   Object depiler();
   boolean estVide();
}
```

-34-

Modélisation



-35-

Type et classe

Pile maPile; *//maPile est de type Pile (interface)*

…

maPile = new ImplementationPile(); *//et de classe ImplementationPile*

..

maPile.empiler(objet); *//quelque soit l'implémentation on en connaît l'interface*

-36-

Héritage «multiple» à la java

```
abstract class C {
        int i;
        void m1( ){...}
        abstract String m2( );
        ...
}
```

```
interface I{
        public final static int j=0;
        int[] p( );
        ...
}
```

```
class C1 extends C  implements I {
        String m2(){...}
        int[] p(){...}
        ...
}
```

-37-

itérateur java: Enumeration

```
public  interface  java.util.Enumeration
{

   public abstract boolean hasMoreElements();
   public abstract Object nextElement();
}
```

-38-

Java: iterateur sur un Vecteur

```
public  class  java.util.Vector     extends  java.lang.Object
                                    implements java.lang.Cloneable  {
   ...
   public Vector();
   public Vector(int  initialCapacity);

   public final void addElement(Object  obj);
   public final boolean contains(Object  elem);
   public final Object elementAt(int  index);

   public final Enumeration elements();

   public final Object firstElement();
   public final boolean isEmpty();
   public final void removeElementAt(int  index);
   public final void setElementAt(Object  obj, int  index);
   public final int size();
}
```

-39-

Java: itérateurs sur une Hashtable

```
public  class  java.util.Hashtable
                                    extends  java.util.Dictionary
                                    implements java.lang.Cloneable  {
   public Hashtable();

   public void clear();
   public Object clone();
   public boolean contains(Object  value);
   public boolean containsKey(Object  key);
   public Enumeration elements();
   public Object get(Object  key);
   public boolean isEmpty();
   public Enumeration keys();
   public Object put(Object  key, Object  value);
   public Object remove(Object  key);
   public int size();
   public String toString();
}
```

-40-

Exemple d'utilisation

```
class Programme{
        ...
        Hashtable baseDeClauses = new Hashtable();  // ensemble indexé de paquets

        public String toString(){
                Enumeration e= baseDeClauses.elements();
                String s="";
                while(e.hasMoreElements()){
                        s += ((Paquet)e.nextElement()).toString()+ rc;}
                return s;
        }
        boolean existePaquet(String s){
                Enumeration e = baseDeClauses.keys();

                while(e.hasMoreElements())
                        if( s==e.nextElement()) return true;
                return false;
        }
...}
```


Classes internes

```
class Pile{
   int taille =0;
   Maillon sommet;
   public void empiler(Object v){...    }
   ....
   public String toString(){...    }

   public Enumeration elements(){
         return new EnumerationPile();
   }
  //------------classe interne----------------------------------------------
  class EnumerationPile implements Enumeration{
         Maillon maillonCourant= sommet;

         public boolean hasMoreElements(){
                 return maillonCourant!= null;
         }
         public Object nextElement(){
                 Object valeurDeRetour = maillonCourant.valeur;
                 maillonCourant=maillonCourant.sousPile;
                 return valeurDeRetour;
         }
   } //-------------------------------------------------------------------------
}
```

Classes internes

peuvent apparaître partout

comme attribut d'une classe, dans un bloc, dans une expression (anonymement)

raison d'être: 'adapter' et 'beans'

simplifie la connection d'objets entre eux

Le nom de classe interne n'est pas utilisable hors de la classe sans préfixer.

Le code des classes internes voit directement les noms englobants.

les classes internes peuvent être déclarées 'static'. Elles sont alors top-level comme les classes du package. L'avantage dans ce cas réside dans la visibilité des attributs et dans la structuration des classes.

-43-

Classes anonymes

```
class Pile{

    ...

    public Enumeration elements(){
        return new Enumeration(){
            Maillon maillonCourant= sommet;

            public boolean hasMoreElements(){
               return maillonCourant!= null;
            }
            public Object nextElement(){
               Object valeurDeRetour = maillonCourant.valeur;
               maillonCourant=maillonCourant.sousPile;
               return valeurDeRetour;
            }
        };
    }
}
```

-44-

Clonage

```
public class Object {
        public final Class  getClass();
        public String  toString();
        public boolean  equals(Object obj);
        protected Object clone()
                                throws CloneNotSupportedException;
        …
}
```

-45-

Cloneable

protected Object clone()

throws CloneNotSupportedException;

Par défaut les objets ne sont pas clonables, cette méthode, dont tout objet hérite, teste donc d'abord si la classe du receveur implémente l'interface Cloneable

public interface Cloneable { }

Si non une exception est levée
Si oui elle crée un objet du même type que le receveur et fait une copie superficielle (affectation des champs)
(les tableaux implémentent cette interface)
La classe Object n'implémente pas elle même cette interface
Si vous spécialisez clone() vous pouvez faire une copie profonde en appelant cette même méthode sur les attributs « clonables »

-46-

Exemple

```
public class Stack implements Cloneable
        {
          private Vector items;
          ….
          protected Object Clone() {//pas obligé de retourner l'exception
            try {
              Stack s = (Stack)super.clone();
              s.items = (Vector)items.clone();
              return s;
            } catch (CloneNotSupportedException e) {
              // Ceci n'arrivera pas car Stack est Cloneable
              throw new InternalError();
            }
          }
        }
```

-47-