Arnaud Strapart
Basé sur le cours LEPL1402

Cours Magistral 5 : Arbres 3a

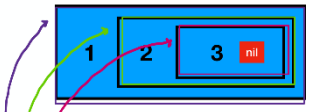
Structure récursive : Les listes (sorte de brocolis) => liste est aussi un cas récursif : cas de base = liste vide = *nil* et cas général : object + liste !! Liste immutable

```
public abstract class RecursiveList<A> implements Iterable<A> {
   public final boolean isEmpty();
   public abstract A head();
   public abstract RecursiveList<A> tail();
   public final RecursiveList<A> cons(final A a);
   public static <A> RecursiveList<A> nil();
}
```

Return the new list [a,this]

Return the nil (empty list) constant

Observe that RecursiveList is an immutable type. Once the object is constructed, the API does not allow to modify it. Interesting no ?




```
public static void main(String[] args) {
   RecursiveList<Integer> l = RecursiveList.nil();
   l = l.cons(3);
   l = l.cons(2);
   l = l.cons(1);
   for (Integer i: l) {
      System.out.println(i);
   }
}
```

Comment implémenter une liste récursive :

```
public abstract class RecursiveList<A> implements Iterable<A> {
   public final boolean isEmpty() { return this instanceof Nil; }
   public abstract A head();
   public abstract RecursiveList<A> tail();
   public final RecursiveList<A> cons(final A a) { return new Cons(a, this); }
   public static <A> RecursiveList<A> nil() { return (Nil<A>) Nil.INSTANCE; }
   public Iterator<A> iterator() { // TODO }
   // LISTE VIDE
   private static final class Nil<A> extends RecursiveList<A> {
      public static final Nil<Object> INSTANCE = new Nil();
      @Override public A head() { throw new NoSuchElementException("empty list"); }
      @Override public RecursiveList<A> tail() { throw new NoSuchElementException("empty list"); }
   }
   // LISTE NON VIDE
   private static final class Cons<A> extends RecursiveList<A> {
      private final A head;
      private final RecursiveList<A> tail;
      Cons(final A head, final RecursiveList<A> tail) {
         this.head = head;
         this.tail = tail;
      }
      public A head() { return head; }
      public RecursiveList<A> tail() { return tail; }
   }
}
```

Impossible de créer d'instance de liste vide

Cannot be instantiated from outside and special static final constant for nil

Cannot be instantiated from outside

Itérator :

Rappel : 2 types (Fail-Safe (stocke tous à l'avance et puis retourner un itérateur sur la collection qu'on ne peut plus changer) et Fail-Fast (lance une erreur car itère la collection tout en la changeant))

Pour la liste récursive : Fail-Safe car immutable

```
public abstract class RecursiveList<A> implements Iterable<A> {

   public Iterator<A> iterator() {
      return new Iterator<A>() {
         // initially current = list on which the iterator is required
         private RecursiveList<A> current = RecursiveList.this;

         public boolean hasNext() {
            return !current.isEmpty();
         }

         public A next() {
            if (current.isEmpty())
               throw new NoSuchElementException("no next");
            else {
               A item = current.head();
               current = current.tail();
               return item;
            }
         }

         public void remove() {
            throw new UnsupportedOperationException();
         }
      };
   }
}
```

This is the outer class instance. The RecursiveList on which iterator() method is called

current.next()

Le current va pointer vers le next

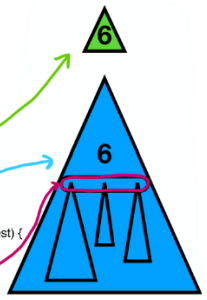
Did you notice that it is implemented as an anonymous class ?

Structure récursive : Les Arbres

Présent partout pour représenter de systèmes de fichiers et expressions arithmétique

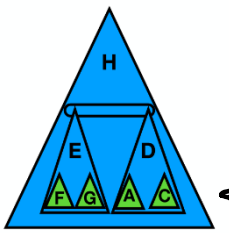
Un arbre est un cas récursif
Cas de base : une feuille avec un élément
Cas général : Racine avec un élément suivit d'une liste de sous arbres

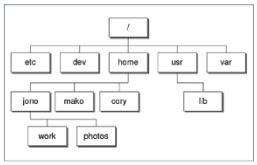
```
public class RecursiveTree<A> implements Iterable<A> {

   public static <A> RecursiveTree<A> leaf(A root) {
      return new RecursiveTree(root,RecursiveList.nil());
   }

   public static <A> RecursiveTree<A> node(A root, RecursiveTree<A> ... subForest) {
      return new RecursiveTree<A>(root,RecursiveList.toList(subForest));
   }

}
```

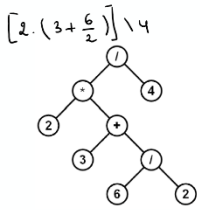


Implémentation d'un arbre

```
public class RecursiveTree<A> implements Iterable<A> {

   private final A root;
   private final RecursiveList<RecursiveTree<A>> subForest;

   private RecursiveTree(A root, RecursiveList<RecursiveTree<A>> subForest) {
      this.root = root;
      this.subForest = subForest;
   }

   public static <A> RecursiveTree<A> leaf(A root) {
      return new RecursiveTree(root,RecursiveList.nil());
   }

   public static <A> RecursiveTree<A> node(A root, RecursiveTree<A> ... subForest) {
      return new RecursiveTree<A>(root,RecursiveList.toList(subForest));
   }

   @Override
   public Iterator<A> iterator() {
      return null; // TODO
   }

   public static void main(String[] args) {
      RecursiveTree tree = node("H",node("E",leaf("F"),leaf("G")),node("D",leaf("A"),leaf("C")));
   }
}
```





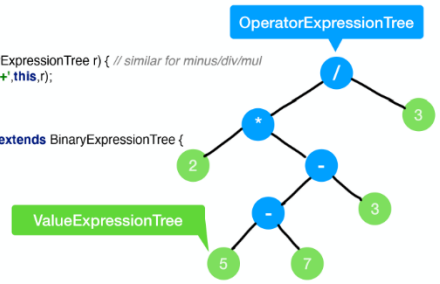



```
public static void main(String[] args) {
   RecursiveTree tree = node("H",node("E",leaf("F"),leaf("G")),
                                 node("D",leaf("A"),leaf("C")));
}
```

Arnaud Strapart
Basé sur le cours LEPL1402

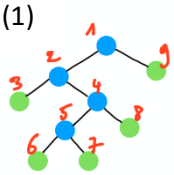
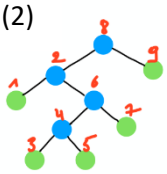
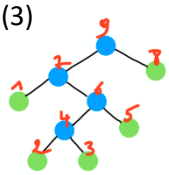
Arbre Binaire = Arbre qui n'a au plus que 2 éléments

```java
public abstract class BinaryExpressionTree {

   private static class OperatorExpressionTree extends BinaryExpressionTree {
      private final BinaryExpressionTree left;
      private final BinaryExpressionTree right;
      private final char operator;

      private OperatorExpressionTree(char op, BinaryExpressionTree l, BinaryExpressionTree r) {
         this.operator = op;
         this.left = l;
         this.right = r;
      }
      public BinaryExpressionTree plus(BinaryExpressionTree r) { // similar for minus/div/mul
         return new OperatorExpressionTree('+',this,r);
      }
   }

   private static class ValueExpressionTree extends BinaryExpressionTree {
      private final int value;

      private ValueExpressionTree(int v) {
         this.value = v;
      }

   }

   public static void main(String[] args) {
      BinaryExpressionTree expr = value(2).mul(value(5).plus(value(7)).minus(value(3)).div(value(3))); // (2 * ((5+7)-3)) / 3
   }
}
```



Avec ce code, on peut afficher l'expression binaire
=> On doit parcourir l'arbre cad récursif (nœud un par un)

!! 3 type d'ordres (donne lieu à des représentation différentes)
(1) Préfixe : Visite puis G puis D
(2) Infixe : G puis visite puis D
(3) Postfixe : G puis D puis visite
!! Doit être capable de donner ordre de visite des nœuds

(1) 
(2) 
(3) 

A quoi ça sert ?

```java
public abstract class BinaryExpressionTree {

   abstract int evaluate();

   private static class OperatorExpressionTree extends BinaryExpressionTree {
      private final BinaryExpressionTree left;
      private final BinaryExpressionTree right;
      private final char operator;

      @Override
      int evaluate() {
         int leftRes = left.evaluate();
         int rightRes = right.evaluate();
         switch (operator) {
            case '+':
               return leftRes + rightRes;
            case '-':
               return leftRes - rightRes;
            case '/':
               return leftRes / rightRes;
            case '*':
               return leftRes * rightRes;
            default:
               throw new IllegalArgumentException("unkown operator " + operator);
         }
      }
   }

   private static class ValueExpressionTree extends BinaryExpressionTree {
      private final int value;

      @Override
      int evaluate() {
         return value;
      }
   }
}
```

```java
public static void main(String[] args) {
   BinaryExpressionTree expr = value(2).mul(value(5).plus(value(7))) // 2*(5+7)
   System.out.println(expr.evaluate());
}
```

Cela permet de donner la valeur de l'expression
Algorithme Postfixe

On peut aussi imprimer l'expression = notation infixe

```java
@Override
public String infixString() {
   return "("+left.infixString()+operator+right.infixString()+")";
}
}

private static class ValueExpressionTree extends BinaryExpressionTree {
   private final int value;

   @Override
   public String infixString() {
      return value+"";
   }

}
}
```

```java
public static void main(String[] args) {
   BinaryExpressionTree expr = value(2).mul(value(5).plus(value(7))) // 2*(5+7)
   System.out.println(expr.infixString());
}
```

Notation postfixe

- (2 * ((5+7)-3)) / 3 => 2 5 7 + 3 - 3 / * 6 = 6 3

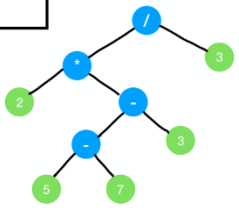
=> pour calculer dans cette notation, on utilise les stacks

Niveau complexité : O(n) => pas possible de faire mieux si on veut imprimer tous les nœuds

Sérialisation : transformer un programme en string (désérialisation string => programme)
Les 3 types d'ordres ne vont pas retranscrire l'arbre unique (uniquement préfix et postfix)

```java
public static void main(String[] args) {
   String postFix = "257+3-/*";

   int l = postFix.length();

   BinaryExpressionTree expr = deserialize(postFix,postFix.length()-1);
   System.out.println(expr.infixString()); // (2 * ((5+7)-3)) / 3
}
```

Serialize
Deserialize



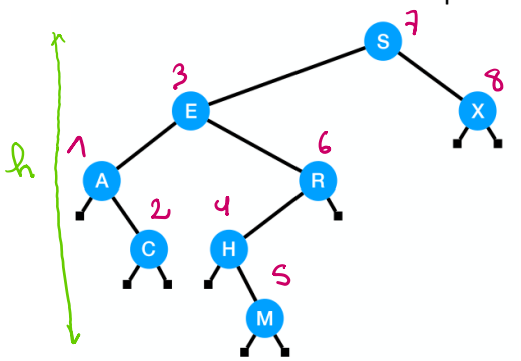
Arnaud Strapart
Basé sur le cours LEPL1402

```java
public static BinaryExpressionTree deserialize(String l, int i) {
    if (Character.isDigit(l.charAt(i))) {
        System.out.println("val:"+l.charAt(i));
        return value(Character.getNumericValue(l.charAt(i)));
    } else {
        char op = l.charAt(i);
        System.out.println("op:"+l.charAt(i));
        BinaryExpressionTree right = deserialize(l,i-1);
        BinaryExpressionTree left = deserialize(l,i-1-right.size());
        return new BinaryExpressionTree.OperatorExpressionTree(op,left,right);
    }
}
```

Désérialisation (par méthode récursive)
=> permet de représenté en infix
String = S_0 S_1 S_2 S_3 .. S_i

Pour sérialisation (📎)

<u>Arbre de recherche (binaire) :</u>
Contiennent des éléments comparables entre eux



Condition pour que ce soit arbre de recherche :
Eléments à gauche > s > Eléments à droite

Dans ordre alphabétique : oui ça fonctionne

Arbre de recherche permet d'énumérer dans ordre croissant (Θ(n)) et pour trouver le plus petit (O(n) ou plus simple O(h))

Pour énumérer les types croissants, quel type de méthode faut-il faire ? Parcours infix

On peut représenter les arbres de recherches avec une structure chainée ou avec un tableau<

Linked BinaryTrees

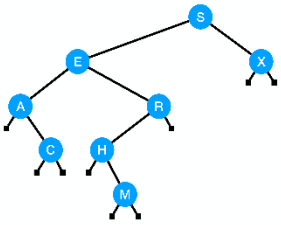
- Linked Structure

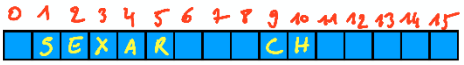
```java
private class Node {
    public int val;
    public Node left;
    public Node right;
    public Node(int val){
        this.val = val;
    }
    public boolean isLeaf(){
        return this.left == null && this.right == null;
    }
}
```

Array-Based BinaryTrees

With an array

- Left = 2*index
- Right = 2*index+1



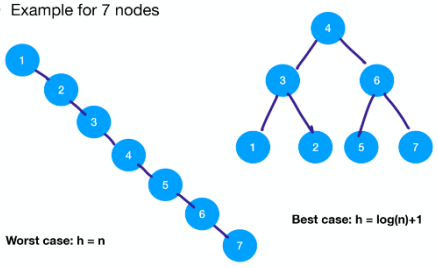


Spatial Complexity ?

Il y a trop de trous !!

Quelle est la relation entre la hauteur de h et le nombre de nœud n ? Si h fixe, alors au max 2^h nœuds

- Example for 7 nodes



Si on recherche le maximum, on va prendre le second car on essaye d'avoir la hauteur la plus petite

Arnaud Strapart
Basé sur le cours LEPL1402

Cours Magistral 6 : Objet Orienté 3b

Le but est de modéliser à l'aide d'objets => **Encapsulation**
Les méthodes dans les classes vont permettre d'insérer des fonctionnalités pour traiter ces objets

!! Il faut toujours mettre les variables d'instance en privé => utilise méthode pour modifier les variables d'instances

Constructeur

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    Bicycle(int initCadence, int initSpeed, int initGear) {
        cadence = initCadence;
        speed = initSpeed;
        gear = initGear;
    }

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp() {
        speed = speed + 5;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" +
                cadence + " speed:" +
                speed + " gear:" + gear);
    }
}
```

Constructor

```
class BicycleDemo {
    public static void main(String[] args) {
        Bicycle bike1 = new Bicycle(3,10,2);
    }
}
```

On peut aussi décider de mettre le constructeur en défaut

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    Bicycle() {
    }

    Bicycle(int initCadence, int initSpeed, int initGear) {
        cadence = initCadence;
        speed = initSpeed;
        gear = initGear;
    }

    void changeCadence(int newValue) {
        cadence = newValue;
```

There is always a default constructor (no parameters). If not explicitly given, an empty one is assumed

Variable d'instance dans privé sont uniques pour chacun des nouveaux objets (différent qd variable d'instance statique)

⇨ On peut créer un objet sans aucun paramètre

Abstraction = cacher les détails inutiles
Plus haut niveau d'abstraction : Interface permet de cacher la manière dont s'est implémenter car ne permet pas d'implémenter les méthodes (contrairement à la classe) !! Classe Abstraite
Ex : Véhicule

```
public interface Vehicle {
   void speedUp();
   void applyBrakes(int decrement);
}
```

A vehicle is a very abstract concept, what does a vehicle look like ?

Héritage = possibilité d'hériter de certains attributs de la classe Parent

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    Bicycle(int initCadence, int initSpeed, int initGear) {
        cadence = initCadence;
        speed = initSpeed;
        gear = initGear;
    }
    void changeCadence(int newValue) { cadence = newValue; }
    void changeGear(int newValue) { gear = newValue; }
    void speedUp(int increment) { speed = speed + increment; }
    void applyBrakes(int decrement) { speed = speed - decrement; }
    void printStates() {
        System.out.println("cadence:" +
                cadence + " speed:" +
                speed + " gear:" + gear);
    }
}
```

Call to Parent's constructor

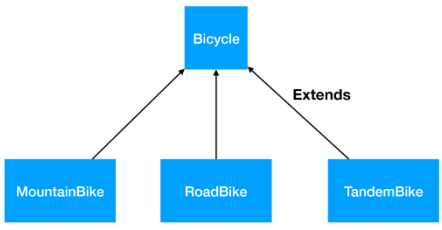
Sub-class

```
class MountainBike extends Bicycle {

    int damping = 0;

    MontainBike(int initCadence, int initSpeed,
                int initGear) {
        super(initCadence,initSpeed,initGear);
    }

    void changeDamping(int newValue) {
        damping = newValue;
    }

    void printStates() {
        super.printStates();
        System.out.println(" damping:" + damping);
    }
}
```

```
class Tandem extends Bicycle {

    boolean freeWheels = false;

    Tandem(int initCadence, int initSpeed,
           int initGear, boolean freeWheels) {
        super(initCadence,initSpeed,initGear);
        this.freeWheels = freeWheels;
    }

    void printStates() {
        super.printStates();
        System.out.println("freewheels:" + freeWheels);
    }
}
```

```
class RoadCycle extends Bicycle {
  // Similar …
}
```

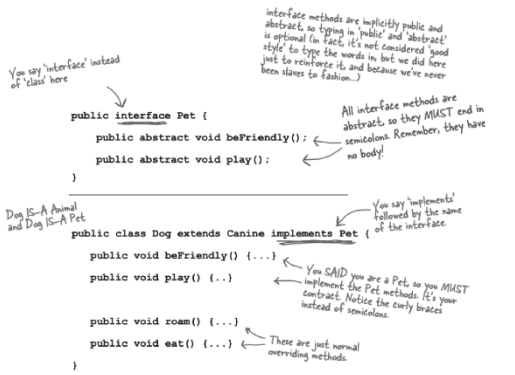
Ex : Tandem, VTT et vélo de route



Mot clé **super** permet d'appeler le constructeur de la classe Parent
Il permet aussi d'appeler une méthode de la classe Parent
=> Programme affiche 120
Qd Puzzle => super va monter dans la hiérarchie jusqu'au moment où il est défini

```
/* Base class vehicle */
class Vehicle {
   int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle {
   int maxSpeed = 180;

   void display() {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}
```

Arnaud Strapart
Basé sur le cours LEPL1402



On ne peut pas avoir plusieurs héritages dans Java (ex : chien => race canine et chien de compagnie)
➔ Problème du Diamant Mortel = si 2 classes Parents ont la même méthode et que dans la classe enfant (qui est lié aux Parents), on demande de renvoyer cette méthode, le programme ne sait pas laquelle choisir !

<= Solution : Interface

Une classe peut implémenter plusieurs interfaces (ex : ArrayList)

<u>Modification des méthodes enfants à partir de la classe Parent</u>
=> On fait un refactoring ds classe Parent

```
void speedUp() {
    if (this instanceof RoadCycle) {
        speed += 10;
    } else {
        speed += 5;
    }
}
```

Mais ce n'est pas ouvert en extension et cela pousse à modifier la classe => or on ne veut plus la modifier

On va utiliser un **@Override** dans classe enfant

```
@Override
void speedUp() {
    speed += 10;
}
```

This is called method Overriding

= <u>Polymorphisme Dynamique</u>
On peut empêcher ce polymorphisme => utilise **final**
!! On peut aussi final une classe c'est-à-dire qu'on ne pourra pas l'étendre

```
class A {
  final void foo() {
    System.out.println("foo A");
  }
}
```

```
final class A {
  void foo() {
    System.out.println("foo A");
  }
}
```

```
class FooA {
  int x = 10;
}

class FooB extends FooA {
  int x = 20;
}

public class PolymorphismMembers {
  public static void main(String args[]) {
    FooA a = new FooB(); // object of type FooB
    System.out.println(a.x);
  }
}
```

Variables are not overriden in Java. So there is no dynamic polymorphism for the variable members of a class

Output 10

Pas de polymorphisme pour les variables d'instance

<u>Délégation</u> = comportement change en fonction du temps (chat meow à GRR)

```
interface SoundBehaviour {

  public void makeSound();
}

class MeowSound implements SoundBehaviour {

  public void makeSound() {
    System.out.println("Meow");
  }
}

class RoarSound implements SoundBehaviour {

  public void makeSound() {
    System.out.println("Roar!!!");
  }
}
```

The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.

```
public class Cat {

  private SoundBehaviour sound = new MeowSound();

  public void makeSound() {
    this.sound.makeSound();
  }

  public void setSoundBehaviour(SoundBehaviour newsound) {
    this.sound = newsound;
  }

  public static void main(String args[]) {
    Cat c = new Cat();
    c.makeSound();        // Output: Meow

    // I would like my cat to sound like a tiger
    c.setSoundBehaviour(new RoarSound());
    c.makeSound();        // Output: Roar!!!
  }
}
```

The sound is delegated to an object

On définit une variable d'instance dans la classe chat qui encapsule son comportement sonore
=> il peut donc maintenant changer le comportement d'un code

Arnaud Strapart
Basé sur le cours LEPL1402

```java
class MultiplyFun {

    static int multiply(int a, int b) {
        return a * b;
    }

    static int multiply(int a, int b, int c) {
        return a * b * c;
    }
}

class Overload {
    public static void main(String[] args) {
        System.out.println(MultiplyFun.multiply(2, 4));
        System.out.println(MultiplyFun.multiply(2, 7, 3));
    }
}
```

Surcharge des méthodes = définir plusieurs méthodes avec le même nom

Mais attention à l'ambiguïté

```java
class MultiplyFun {

    static long multiply(long a, int b) {
        return a * b;
    }

    static long multiply(int a, long b) {
        return a * b;
    }

}

class Overload {
    public static void main(String[] args) {
        System.out.println(MultiplyFun.multiply(2, 4));
    }
}
```

Compiler will complain because it is ambiguous which method should be executed.

Les constructeurs peuvent s'appeler entre eux avec le mot **this()** grâce au surchargement

```java
public class Duck {

    String name;
    boolean canFly;
    int maxSpeed;

    public Duck() {
        this.name = "generic";
        this.canFly = false;
        this.maxSpeed = 10;
    }
```

```java
public Duck(String name) {
    this();
    this.name = name;
}
```

```java
public Duck(String name, boolean canFly) {
    this(name);
    this.canFly = canFly;
}

}
```

Polymorphisme Statique = Overload des méthodes

```java
class A {
}
class B extends A { }
class C extends B { }
class D extends C { }
class Test {

    public static void foo(A a) {
        System.out.println("foo a");
    }
    public static void foo(B b) {
        System.out.println("foo b");
    }
    public static void foo(D d) {
        System.out.println("foo d");
    }
}
```

```java
public static void main(String args[]) {
    A d = new D();
    System.out.println(d.getClass());
    foo(d);
}
```

Do you see "foo a" or "foo d" ?

```java
public static void main(String args[]) {
    C d = new D();
    System.out.println(d.getClass());
    foo(d);
}
```

Do you see "foo a" or "foo b" or "foo d" or it doesn't compile ?

((1)) car c'est un type A => foo(a)
((2)) C n'a pas de fonction foo(), on regarde alors en B => foo(b)

4 principes d'orientation d'objet :
1) Pas de copier-coller
2) Encapsuler un maximum cad mettre des **Private**
3) 1 seule responsabilité par classe (ex canif qui ouvre tous) => pas d'héritage possible / pas modulaire
4) Ouvert en extension et fermé en modification (ex : le tri des object comparateur ou bruit du chat)
➔ Outil pour faire ça : Polymorphisme et Délégation