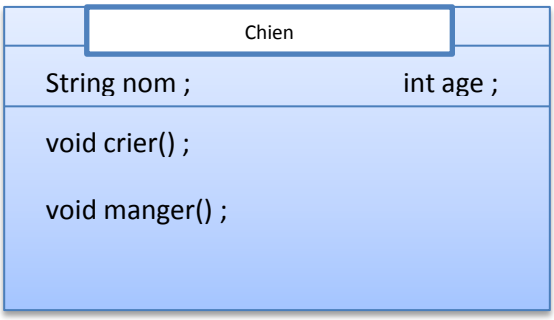
3TC- ELP : FICHES

JAVA

Compilation/Exécution

Compilation : **javac HelloWorldApp.java**
Exécution : **java HelloWorldApp**

Classes et objets



Constructeur : **public Chien(){…}**
Surcharge de constructeur : **public Chien(String myName, int myAge){…}**
Appel de méthode : **monChien.crier() ;**
Le mot-clef **static** convient pour les :

- *méthodes* indépendantes de tout état interne et ne pouvant être implémentées que d'une seule manière (Math.cos()).
- *attributs* dont les valeurs sont partagés par tous les objets d'une même classe:
    - compteur d'instances,
    - constantes (Math.PI). Dans ce cas on fait suivre le mot-clef **final** à **static**.

/ !\ Les attributs ont tous des valeurs par défaut.
Le mot-clé **this** permet de différencier des attributs de variables locales, ou d'appeler un constructeur : **this(hautGauche, new Point(longEnX, longEnY));**
La *composition* est la relation entre objets dans laquelle un objet est l'attribut du second.

Tableaux

int myTab[5] ; //Taille : **myTab.length**

Utilité du mot-clé private

- Les méthodes **private** ne sont (normalement) d'aucune utilité pour le client.
- Si le client les utilise mal, il pourrait rendre l'état interne de l'objet incohérent.
- Si le client les utilise et que leur implémentation change, son code devra changer aussi.

Interfaces

Les *interfaces* comprennent les signatures des méthodes publiques de certains objets. Des objets différents peuvent avoir les mêmes méthodes, par exemple les objets Chien et les objets Chat qui implémenteraient l'interface Animal (ils sont de *type* Animal) avec les méthodes **void crier() ;** et **void manger() ;**.
Le *polymorphisme* désigne le fait que tous les objets du même type sont interchangeables.
La classe *dérivée* hérite des attributs et méthodes publiques de la classe-mère et y ajoute de méthodes propres à elle (Capitale dérive de Ville). C'est le principe de *l'extension*, suggéré par le mot-clef **extends**.
Un interface peut hériter de plusieurs interfaces.
Pour appeler le constructeur de la classe de base : **super() ;**

Mot-clé abstract

Méthode abstraite -> Pas de corps.
Classe avec au moins 1 méthode abstraite -> Mot-clé **abstract** dans la déclaration -> Ne peut pas être instanciée.
La classe dérivée redéfinit les méthodes (abstraites notamment).
/ !\ *Redéfinition* : même signature, mais corps différents entre la classe de base et la classe dérivée.
Surcharge : même nom, mais liste de paramètres différente, au sein d'une même classe.

ANT - Fichier JAR

Fichier .jar : Archive compressée de classes JAVA, téléchargeable en une fois et exécutable, signable par son auteur.
Ant : Outil pour compiler automatiquement (commande shell ant) réalisant des tâches listées dans build.xml .
On utilise **package** pour regrouper des classes.
La JVM utilise le *classpath* pour savoir où chercher les fichiers en .class .

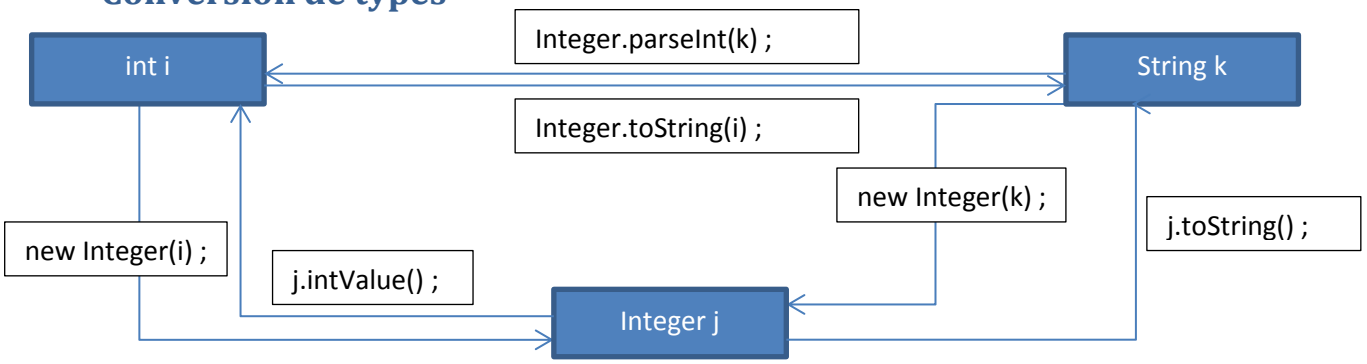
Exceptions

Bloc **try{…}catch(Exception e){…}finally{…}** où **finally** (optionnel) s'exécute toujours.
/!\ **import java.lang.Exception;**
Le développeur peut indiquer qu'une méthode est susceptible de lever une exception avec **throws** et peut la lever avec **throw**.

```
public int plusDeux() throws ExceptionValue{
  if(this.value==null)
  { throw new ExceptionValue("valeur nulle"); }
}
```

On peut créer la classe ExceptionValue avec **public class ExceptionValue extends Exception {…}**.

Conversion de types



Collections

- **List** autorise les clones, pas **Set**. **Map** a deux objets par entrée, une clé et une valeur, et peut contenir plusieurs fois la même valeur mais jamais plusieurs fois la même clé. Une **List** est ordonnée, pas un **Set**. Un **Map** s'assimile à une table de hachage.
- Un **ArrayList** s'assimile à un tableau, on peut accéder à la donnée avec son indice ; avec **LinkedList** on ne peut pas, il faut la parcourir jusqu'à la case qu'on veut.
- **TreeMap** conserve l'ordre des clés, pas **HashMap**. **TreeSet** correspond à un objet, **TreeMap** à deux objets (clé et valeur). **TreeSet** est implémenté grâce à un **Set** (un ensemble), **TreeMap** grâce à un **Map** : il peut y avoir des valeurs dupliquées dans **TreeMap** mais pas dans **TreeSet**.

/ !\ **import java.util.nomCollection ;**

Flux

System.out : Application/Ecran pour une info
System.err : Application/Ecran pour une erreur
System.in : Clavier/Application
/ !\ **import java.io ;**
Flux d'entrées :

- **abstract int read()** Lit le prochain octet. La valeur de l'octet est retournée comme un entier entre 0 et 255, -1 si la fin de fichier est atteinte.
- **int read(byte[] b)** Lit un certain nombre d'octets et les copie dans le buffer b. Le nombre d'octets lu est retourné, -1 si la fin de fichier est atteinte.
- **void close()** Ferme proprement le flux.

Flux de sorties :

- **void write(int b)** Ecrit l'octet b (seulement les 8 bits de poids faibles sont pris en compte).
- **void write(byte[] t)** Ecrit les octets du tableau d'octets t.
- **void close()** Ferme proprement le flux.

```
File f = new File("source.txt");
BufferedReader fin = new BufferedReader( new FileReader(f) );
String line = fin.readLine(); //On lit une ligne de source.txt
PrintWriter fout = new PrintWriter( new FileWriter(f) );
fout.println("blabla"); //On écrit blabla dans source.txt
```

La *sérialisation* est le processus de transformation d'un objet en flux d'octets : **java.io.Serializable**.

Threads

```
public classe MaClasseRunnable implements Runnable {
  public void run() { ... }
}
```

Pour démarrer le thread :

```
MaClasseRunnable r = new MaClasseRunnable();
new Thread(r).start();
```

Méthodes : start, interrupt, isAlive, sleep, currentThread, join…
Mot-clé **synchronized** pour éviter les interferences/incohérences.
join : attend que le thread dont on appelle la méthode **join** meurt.
wait : met en attente le thread courant, jusqu'à ce qu'il soit réveillé/interrompu par un autre thread.
notify: réveille le thread qui est en attente sur l'objet dont on appelle la méthode **notify**.
notifyAll: réveille tous les threads en attente sur l'objet.

GO

Histoire & Objectifs

- Programmation système
- Applications & Cloud-Computing
- Langage structurellement et statistiquement typé

Hello World

```
package main  //Exécutable
import ( //Packages à importer
  "fmt"
)
func main() { //Fonction
  fmt.Println("Hello, world!")
}
```

Compilation : **go build**
Exécution : **go run hello-world.go**

Fonctions & Variables

```
type intFunc func(int) int //Déclaration de la fonction retournée
func Adder(x int) intFunc {
  return func(y int) int {
    return x + y
  }
}
```

Les chaînes de caractères sont des **string**. La longueur d'une chaîne est **len(s)**, et le i-eme élément est **s[i]** (0 <= i < len(s)). Un booléen est de type **bool** et a pour valeurs **true** ou **false**.

Tableaux & Slices

Un tableau possède une *taille* et une *capacité* ; ses vues (parties du tableau) sont des *slices*.
Tableau de type T : []T

```
x :=[]int{1,2,3}
x=append(x, 4, 5, 6) //x=[ 1 2 3 4 5 6]
y := make([]int, 5, 10) //y=[0 0 0 0 0] avec une longueur de 5 et une capacité de 10
copy(y, x) //Remplace le contenu de y par celui de x
dump(x[1 :]) //Affiche le contenu de x à partir de l'indice 1
```

Map & Struct

Une Map est un tableau possédant des clés permettant d'accéder aux valeurs.

```
maMap := make(map[string]int) //Clé : String, Valeur : int
delete(ints, "4") //Efface la valeur et la clé correspondant à la clé « 4 »

type Person struct {
  name string
  email string
```

```
  age uint8
}
mrbean := Person{"Mr Bean", "bean@outlook.com", 59}
fmt.Println(mrbean)
fmt.Println(mrbean.email)
```
Person est visible en dehors de son module de définition car le nom commence par une majuscule, name, email et age ne sont accessibles que pour les fonctions du module de définition, ce qui ne serait pas le cas si leurs noms avaient été Name, Email et Age.
```
var rowan *Person = new(Person)
rowan.name = "Rowan Atkinson"
rowan.email = rowan.atkinson@outlook.com
rowan.age = 59 fmt.Println(rowan)
```

Structures de contrôle

```
if err := file.Chmod(0777); err != nil {
  // ... } //';' équivaut à un AND dans les conditions
a := []int{0, 1, 2, 3, 4}
for i := range a {
  // ...
}
ints := map[string]int{ "1": 1, "2": 2, "3": 3 }
for key, value := range ints {
  // ... }
for c := range "Hello world" {
  // ... }
```
Les switch/case existent aussi en Go.

Channels & Goroutines

Une Goroutine est une fonction s'exécutant dans un processus distinct léger. Une *channel* permet de synchroniser producteur et lecteur.
```
func main() {
  stop := make(chan int)    //Création d'une channel
  go func() {
    fmt.Println("I am in a goroutine")
    stop<- 0
  }()
  <-stop
  fmt.Println("Bye-bye“)
}
```

Multiplexage avec Select

`select` permet de faire du multiplexage de channels en prenant de façon équitable une valeur dans un des channels manipulés. Le package time offre plusieurs fonctions donnant des channels, comme `After` et `Tick`. De même, `os/signal` donnne des channels pour traiter des signaux Unix.
L'instruction `defer` permet d'exécuter une instruction à la fin d'une fonction, peut importe quelle branche fera le `return`. C'est par exemple utile pour s'assurer qu'un fichier sera fermé, sans devoir penser à le faire pour chaque branche possible.
Une interface liste les méthodes qu'un type doit implémenter pour *satisfaire* l'interface.

Clojure

Histoire & Objectifs

- Optionnellement typé
- Possibilité d'y intégrer du Java
- Big data, traitement de données
- Notation polonaise inversée : 1+2 -> **(+ 1 2)**

Variables & Fonctions

(def a 10) Pour nommer une valeur

```
(def twice
  (fn [x] (* 2 x))) //Fonction anonyme
(twice a)

(defn maximum //Fonction nommée
  ([x] x ) //Arguments : x
  ([x y] (if (> x y) x y) )) //Arguments : x et y
(maximum 1)
(maximum 1 2)
```

Structures conditionnelles

```
cond
   (= x 1) "one"
   (= x 2) "two"
   (> 100) "at least hundred"
   :else   "something"
```

Listes

Définition d'une liste : **(def my-list '(1 2 3 4 5 6))**
(cons 100 my-list) //my-list (100 1 2 3 4 5 6) l'original n'est pas modifié
(conj my-list 0) // (0 1 2 3 4 5 6) l'original n'est pas modifié

Filter, Map & Reduce

Filter filtre un élément selon un prédicat :
(filter even? ; [1 2 3 4 5 6 7 8 9 10]) // Donne (2 4 6 8 10)
Map applique une fonction à chaque élément d'une collection : **(map (fn [x] (+ 10 x)) '(1 2 3 4)) ; Donne (11 12 13 14)**
Reduce permet de réduire une collection à une seule valeur.

JavaScript

Histoire & Objectifs

- Dynamisation des pages HTML avec l'utilisateur
- Faiblement typé
- Pas de compilation : node pour l'interpréteur interactif, node toto.js pour lancer le script toto

Variables

Type déclaré en fonction de l'usage de la variable.
Déclaration : **var x**
Affichage : **console.log(« coucou » + x)**

Structures de contrôle

- if / else if / else
- while ()
- do {} while ()
- for () / break
- switch() / case / break

Fonctions

```
function somme(a, b) { //On ne donne pas le retour, ni le type des paramètres
  return a+b
}
```

/ !\ Une variable peut être de type fonction, et une fonction peut être passée en paramètre à une autre fonction ou passée comme retour de fonction, comme en GO.
/ !\ Pas de règles sur le nombre de paramètres d'une fonction

```
var chose = { "hello" : "coucou", 3:10}
```

- ➔ Clés : « hello » et 3
- ➔ Valeurs : « coucou » et 10

```
tesAmis = new Array();
tesAmis.push(1); //On peut rajouter des cases à un tableau
tesAmis.push('leon'); //Le tableau contient n'importe quel type
```

nomFonction.length : Nombre d'arguments passés en paramètres

Map & Reduce

Map : Fabrique un nouveau tableau à partir de l'application d'une fonction sur tous les éléments du tableau
Reduce : Réduit le tableau à une valeur unique

Objets

```
function parle (phrase) { //Méthode
  console.log("Le lapin ", this.couleur, " dit '", phrase, "'");
}
var lapinBlanc = { couleur : "blanc", parle : parle }; //Initialisation des attributs
var lapinNoir = { couleur : "noir", parle : parle };
lapinBlanc.parle(" Je suis tout blanc ");
lapinNoir.parle(" Je suis tout noir "); //Application méthode
```

Ruby

Exécution et Hello World

puts "Hello, world!"
Exécution : **ruby hello.rb**

Spécificité

Tout est objet, on ne diffère pas les entiers des chaînes de caractères ou autres.
La valeur de retour d'une fonction est celle de la dernière expression évaluée.
? : fonctions booléennes.
! : fonctions qui ont des effets de bord notables.
Une fonction sans paramètres peut se déclarer et s'invoquer sans les parenthèses.

Classes

```
class Calc
  def initialize(init = 0) //Constructeur
    @last = init //Attribut
  end
  def add(n) //Méthode de classe
    @last = @last + n
    self //Equivalent à 'this' en JAVA
  end
  # Ceci est un commentaire
  c = Calc.new(10)
  puts "c = #{c.add(1)}"
```

Array & Hash

```
a = [1, 2, 3]
a[0]
h = { "Foo" => "Bar", :plop => 10 }
h[:plop]
```

Fonctions anonymes

```
# Adapté aux formes courtes
{ |a, b, c| code }
# Adapté à du code plus long
do |a, b, c|
  code
end
```

Structures de contrôle

```
if cond then a() elseif othercode then b() else c() end
```

```
result = case n when 10 then "ten" when 20 then "twenty" when 30..100 then "between 30 and 100" else "something else" end
while cond a() b() break if c() end
```

JSON

```
require 'json'
puts JSON.generate({
  :id => 69,
  :name => "Mr Bean",
  :age => 58,
  :friends => [3223876, 73635, 9912983]
})
```