Programmation avancée

Historiques des paradigmes

Impérative: un des premiers paradigmes. Les programmes sont considérés comme une suite linéaire d'instructions, exécutées l'une après l'autre de façon séquentielle. Très mécanique et intuitif parce que basé sur la causalité directe. C, Pascal, Cobol, Assembleur ou Fortran
POO: basé sur l'impératif, programmation centrée autour du concept d'objet. La POO permet d'utiliser les notions d'encapsulation, d'héritage et de polymorphisme. Polymorphisme: interface unique pour des entités pouvant être de différents types. Généricité: définition d'algorithmes identiques opérant sur des types différents. Java, C++, Javascript, Kotlin, C# ou Python
Déclarative: apparaît après l'impérative, consiste à définir le résultat final voulu, c'est au système de déterminer la marche à suivre pour parvenir à ce résultat. On se concentre sur le quoi plus que sur le comment, ce qui produit du code expressif et concis. Offre de la flexibilité comme le compilateur/interpréteur peut gérer de différentes manières ce qui conduit à des optimisations. Lisp, SQL, ou Haskell
Fonctionnelle: dérive de la déclarative, centré autour de la notion de fonction pure, une fonction qui avec les mêmes paramètres d'entrée renvoie toujours le même résultat. Limite au maximum les effets de bords. Code prévisible, limite les conflits d'accès concurrentiels. Lisp, OCaml, Erlang ou Scala
Logique: dérive de la déclarative, programmation conduite par le raisonnement, par le biais de formule de déclaration (prédictas). Les prédicats forment la base de connaissance utilisée par le système pour déduire des réponses. Les programmes ressemblent plus à des collections de déclarations qu'à des séquences d'instructions. Prolog, Oz ou CLIPS
Réactive: dérivant du pattern observer, la programmation réactive fonctionne en propageant les modifications d'une source réactive à des éléments dépendants de cette source. Principalement présent dans les interfaces graphiques pour permettre une interaction avec l'utilisateur. RxJS, RxJava, RxScala ou RxSwift

Impérative

Ben impératif quoi...

POO

Rester le plus possible le plus abstrait possible, faire le plus de liens possibles le plus haut dans la hiérarchie et concrétiser le plus tard possible

Enumération: définition d'un ensemble fini de constantes. Nom des constantes en majuscules (convention)
Interface: ensemble de méthodes abstraites et éventuellement de constantes. Permet de définir le comportement d'une classe, obligatoirement implémenté par celle-ci, sans l'implémenter au niveau de l'interface. L'interface défini un contrat à respecter par les classes qui implémentent cette interface. Une classe peut implémenter plusieurs interfaces, une interface peut être implémentée par plusieurs classes, sans notion de hiérarchie. L'interface étant abstraite par définition, inutile de préciser que les méthodes dedans le sont
Classe abstraite: une classe abstraite ne s'instancie pas et est destinée à l'héritage. Elle peut contenir des méthodes abstraites, ainsi que des attributs. Les méthodes abstraites doivent être redéfinies dans les classes concrètes

```
public enum Atout {
	// Il faut être en UTF-8 avec Eclipse (preference-general-content type-text / projet-propriétés-ressources )
```

```
	PIQUE("♠"), COEUR("♥"), CARREAU("♦"), TREFLE("♣"); //PIQUE("PIK"), COEUR("COE"),
CARREAU("CAR"), TREFLE("TFL");

	private final String visuel;

	private Atout(String visuel) {
		this.visuel = visuel;
	}

	@Override public String toString() {
		return visuel;
	}
}

public interface ICarte {
	public static final Dimension DIMENSION = new Dimension(59,91);

	public void tourner();

	public default boolean estSuperieureOuEgale(ICarte carte) {
		this.tourner();
		return false;
	}
}

public abstract class Carte implements ICarte {
	private final String dos;
	private boolean visible=false;

	protected abstract String getFace();

	public Carte(String dos) {
		super();
		this.dos = dos;
	}

	@Override
	public void tourner() {
		visible = ! visible;
	}

	@Override
	public String toString() {
		String rep;
		if (!visible) {
			rep=dos;
		}
		else {
			rep=this.getFace();
		}
		return rep;
	}
}

public abstract class CarteClassique extends Carte {
	private final Valeur valeur;
	private final Atout atout;
```

```
	public CarteClassique(String dos, Valeur valeur, Atout atout) {
		super(dos);
		this.valeur = valeur;
		this.atout = atout;
	}

	/**
	 * Carte32 c1 = new Carte32(...);
	 * Carte32 c2 = new Carte32(...);
	 * c1.estSuperieureOuEgale(c2);
		 */
	@Override
	public boolean estSuperieureOuEgale(ICarte carte) {
		boolean rep=false;
		try {
			CarteClassique carteClassique = (CarteClassique) carte;
			rep=this.valeur.ordinal()>=carteClassique.valeur.ordinal();
		} catch (ClassCastException e) {
			System.out.println("TODO DEBUG : problème de cast , ICarte comparée n'est pas une
Carte Classique");
		}
		return rep;
	}

	@Override
	protected String getFace() {
		return "["+this.valeur.toString()+" "+this.atout.toString()+"]";
	}
}
public class Carte32 extends CarteClassique {
	private static final String DOS_CARTE="[****]";

	public Carte32(Valeur valeurCarte, Atout couleurCarte) {
		super(DOS_CARTE, valeurCarte, couleurCarte);
		// vérification simple de la valeur
		if (valeurCarte.ordinal()<Valeur.SEPT.ordinal()) {
			System.out.println("PROBLEME Carte 32, Valeur non autorisée");
		}
	}
}
public abstract class Paquet<T extends ICarte> implements IPaquet<T>{
	/*
	 * Patron de conception façade : on agrège une instance de List<T> plutôt que d'implémenter
	 * List<T>. On conserve également la possibilité de faire un extends d'une de nos propres
	 * classes et on ne redéfinit que les méthodes qui nous intéressent. Nous ne sommes pas
	 * obligés d'implémenter toute l'interface List<T>
	 */
	protected List<T> cartes;
	protected abstract void creerPaquet();

	public Paquet() {
		super();
		this.creerPaquet();
		this.shuffle();
```

```
	}

	/*
	 * façade : redéfinition de la méthode size
	 */
	public int size() {
		return this.cartes.size();
	}

	/*
	 * façade : redéfinition de la méthode add qui retourne un booléen toujours true
	 */
	public boolean add(T uneCarte) {
		return cartes.add(uneCarte);
	}

	/*
	 * façade : redéfinition de la méthode get
	 */
	public T get(int i) {
		T carte = cartes.get(i);
		return carte;
	}

	protected String toString(int nbColonnes) {
		String rep="";
		for (int i = 0; i < this.size(); i++) {
			T elem = cartes.get(i);
			rep+=elem.toString();
			if (i%nbColonnes==(nbColonnes-1)) {
				rep+="\n";
			}
		}
		return rep;
	}

	@Override
	public String toString() {
		return this.toString(5);
	}

	public void shuffle() {
		Collections.shuffle(cartes);
	}
}

public class Paquet32 extends Paquet<Carte32> {
	// instance privée agrégée pour le singleton
	private static Paquet32 instance = null;

	// méthode publique getInstance pour le singleton
	public static Paquet32 getInstance() {
		if (instance == null) {
			instance = new Paquet32();
		}
		return instance;
```

```
	}

	//constructeur privé pour le singleton
	private Paquet32() {
		super();
	}

	protected void creerPaquet() {
		super.cartes= new ArrayList<Carte32>(32);
		for (Atout atout : Atout.values()) {
			for (int i = Valeur.SEPT.ordinal();
					i <= Valeur.AS.ordinal(); i++) {
				Valeur valeur = Valeur.get(i);
				super.add(new Carte32(valeur, atout));
			}
		}
	}

	@Override
	public String toString() {
		return super.toString(4);
	}
}
```

Etre vigilant sur les constructeurs, la visibilité, les setters, les getters, @Override... Ne pas hésiter à commenter

Patrons de conception POO (design patterns):
Singleton: garanti que l'instance d'une classe n'existe qu'en un seul exemplaire et fourni un point d'accès global à cette instance (voir "Paquet32")

Facade: procure une interface qui offre un accès simplifiée à une librairie, un framework, un ensemble complexe de classes, ou un objet (voir "Paquet")

Déclarative

3 conditions permettant de faire une fonction récursive:

- Condition d'arrêt
- Résolution du problème
- Appel récursif

Fonctionnelle

Syntaxe proche OCaml
Indique si deux listes sont égales

```
equal:
[], [] : true
[], t::q : false
t::q, [] : false
t::q, t1::q1 : t == t1 && equal(q, q1)
```

Indique si la liste est triée

```
sorted:
[] : true
[t] : true
t1::t2::q : t1 < t2 && sorted(t2::q)
```

Fonction arithmétiques

```
add = lambda x, y: x + y
substract = lambda x, y: x - y
multiply = lambda x, y: x * y
divide = lambda x, y: x / y #3)
```

Si c'est pair et si c'est multiple de 42

```
isPair = lambda x: x % 2 == 0
isMultipleOf42 = lambda x: x % 42 == 0
```

Vérifie une condition donnée par un prédicat pour un intervalle de chiffres donné

```
def returnNumbers(min, max, predicat):
	return [x for x in range(min, max) if predicat(x)]

print("ReturnNumbers", returnNumbers(3, 37, lambda x: x % 4  == 0))
```

Compte le nombre de voyelles dans un texte

```
def nbVoyelles(text):
	VOYELLES_FR = "AaÀàEeÈèÉéIiOoUuÙùYy"
	return len([x for x in text if x in VOYELLES_FR])
```

Fonction prenant une fonction en paramètre et renvoyant une fonction qui s'applique 3 fois

```
def f(func):
	return lambda x, y: func(func(x, y), func(x, y:)

def multiply(a, b):
	return a * b

print("multiply", f(multiply)(2, 4)) # 2 x 4 x 2 x 4 renvoie 64
```

Lors de la déclaration d'une fonction, elle crée un espace fermé de valeurs auxquelles elle a accès. Une fonction déclarée dans une autre fonction inclut dans sa fermeture (closure) les paramètres et variables de la fonction englobante
Avec une fermeture, permet de récupérer une fonction qui divise les nombre d'une liste par le nombre indiqué

```
def diviseListePar(divider):
	if divider == 0:
		raise NameError('Division by zero')
	return lambda listToDivide: [x / divider for x in listToDivide]

diviseListePar2 = diviseListePar(2)
print(diviseListePar2(range(20)))
```

Avec une fermeture, permet de récupérer une fonction qui vérifie le nombre d'éléments dans un liste, type d'élément indiqué dans la fermeture

```
def compterNombre(value):
	return lambda listToSearch: len([x for x in listToSearch if x == value])
```

```
compterNombre2 = compterNombre(2)
print(compterNombre2([0,1,2,3,4,5,4,3,2,1,0])==2)
```

Compréhensions listes python:
Syntaxe: nouvelle_liste = [traitement boucle_for test_optionnel]

```
string_numbers = [str(elt) for elt in my_list if type(elt) == int]
```

Si il n'y a pas de traitement sur les éléments retenus, on se contente de mentionner la variable à ajouter dans la nouvelle liste, elt dans l'exemple ci-dessus

Logique

Les variables commencent par une majuscule ou un underscore. Une structure se compose d'un foncteur et d'une suite d'arguments. Un atome est une string commencant par une minuscule. C'est une constante comme les chiffres
Nombre pair

```
pair(0).
pair(X) :-
        X>0,
        X2 is X-2,
        pair(X2).
```

Trouve la somme des N premiers entiers

```
som(0,0).
som(N,X) :-
        N>0,
        N1 is N-1,
        som(N1,X1),
        X is N+X1.

som(4, X).
```

Donne "X = 10"

Factorielle d'un nombre

```
fact(0,1).
fact(N,X) :-
        N>0,
        N1 is N-1,
        fact(N1,X1),
        X is N*X1.
```

Fibonacci

```
fibo(1,1).
fibo(2,1).
fibo(N,X) :-
        N>2,
        U is N-1,
        V is N-2,
        fibo(U,U1),
        fibo(V,V1),
        X is U1+V1.
```

On défini les hommes, les femmes "femme(anna)", les parents "parent(annette, anna)"
Règles de famille

```
pere(X,Y):- parent(X,Y),homme(X).
mere(X,Y):- parent(X,Y),femme(X).
enfant(A, P) :- pere(P, A) | mere(P, A).
enfant(X) :- enfant(X, _).
freresoeur(X,Y) :- pere(P,X),pere(P,Y),mere(M,X),mere(M,Y), X==Y.
frere(X,Y) :- freresoeur(X,Y),homme(X).
soeur(X,Y) :- freresoeur(X,Y),femme(X).
oncletante(X,Y) :- parent(Z,Y),freresoeur(X,Z).
oncle(X,Y) :- oncletante(X,Y),homme(X).
tante(X,Y) :- oncletante(X,Y),femme(X).
cousincousine(X,Y) :- oncletante(Z,X),enfant(Y,Z).
cousin(X,Y) :- cousincousine(X,Y),homme(X).
cousine(X,Y) :- cousincousine(X,Y),femme(X).
```

Prédicat d'appartenance (liste)

```
appartient(E,[E|_]).
appartient(E,[X|Q]):-
        E==X,
        appartient(E,Q).
```

Opérateur égalité: numérique =:= littéral ==
Opérateur inégalité: numérique == littéral ==
Opérateur plus petit: numérique < littéral @<
Opérateur plus petit ou égal: numérique =< littéral @=<

Attention aux majuscules, aux points, aux espaces, aux backslashs!!

Réactive

Observable rudimentaire

```
function anObservable(observer)
        for (let i=1; i<11; i++) {
                setTimeout(() => observer(i + 40), i * 500);
        }
}

anObservable(data => console.log(data));
```

Idéalement un observable prend en callback une fonction pour next (une valeur est poussée), error (une erreur est poussée) et complete (l'observable n'émettra plus de valeurs)