M2 ISF

Introduction à la P.O.O et langage java (fascicule 1)

O. Auzende
2012-2013

Programmation Orientée Objet et langage java

Objectifs

Comprendre les principes de la programmation orientée objet : notions de classe, instance, héritage, polymorphisme. Les appliquer en langage java.

Sommaire / Plan

Introduction à la P.O.O
Classes et instances
Structures de contrôle
Héritage et polymorphisme
Les paquetages
Le paquetage graphique
La gestion des événements : méthode MVC
Les conversions et calculs
Applets java
Programmation d'une application financière
Transformation de l'application financière ensuite en applet.

Logiciels utilisés

JDK version >= 1.6 et sa documentation, téléchargeables librement sur le site de Sun

Pré-requis

Avoir programmé dans un langage informatique.

Bibliographie

Programmer en java, java 5.0, Claude DELANNOY, Editions Eyrolles.
Java et la programmation orientée objet, Michel DIVAY, Editions Dunod
La programmation Orientée Objet, Hugues BERSINI, Editions Eyrolles
JAVA 6, Les fondamentaux du langage Java, Thierry GROUSSARD, Collection Ressources Informatiques
Développement JEE 5 avec Eclipse Europa, Karim DJAAFAR, Editions Eyrolles

Programmation Orientée Objet et langage java

Introduction

Langages procéduraux : C, Pascal, Basic, Fortran, Cobol… Un programme est une suite d'instructions, les données étant stockées dans des **tableaux** (tableaux de nombres, tableaux de chaînes de caractères…).

Apparition de l'**intelligence artificielle** : structuration des informations en **objets conceptuels** ; les données de types variés (nombres, chaînes de caractères) concernant **un même objet** sont regroupées en mémoire. Un objet est donc **composite**.

Exemple :
- compte en banque = ensemble d'informations (numéro, titulaire, opérations, solde)
- titulaire = ensemble d'information (nom, prénom, adresse, tel, mail)
- opération = ensemble d'informations (date, lieu, versement ou retrait, montant)

→ Apparition de la **Programmation Orientée Objet** (P.O.O) et des **langages à objets** :

Nom	Langage à objets	SE	Remarques
Simula (1967)	Introduction de la notion de classe	Unix	
Smalltalk (1976)	OUI	Unix / Windows	Environnement complet (SE compris)
Eiffel (1986)	OUI	Spécifique	Environnement complet
Objective C (1986)	OUI	Macintosh	Basé sur Smalltalk
C++ (années 1990)	MIXTE	Unix	Le plus difficile et le plus dangereux : procédural + objets
java (1995)	**OUI**	**TOUS**	**Plus accessible que C++, et surtout multi-plateforme**
C# (2001)	MIXTE	Windows	

Avantages de la P.O.O. : amélioration de la spécification des besoins, prototypage rapide, maintenance plus facile, réutilisabilité, portabilité.

Classes et instances

Création d'objets à partir de classes : attributs d'instances

Les données sont structurées en **objets**, composés d'**attributs**. Chaque objet a sa propre identité : deux objets dont les attributs ont les mêmes valeurs sont distincts (vu qu'ils n'ont pas le même **emplacement mémoire**).

Un objet est nécessairement créé à partir d'un modèle (un « moule ») appelé une **classe**. Il est alors appelé une **instance de la classe**, et ses attributs sont appelés des **attributs d'instance**.

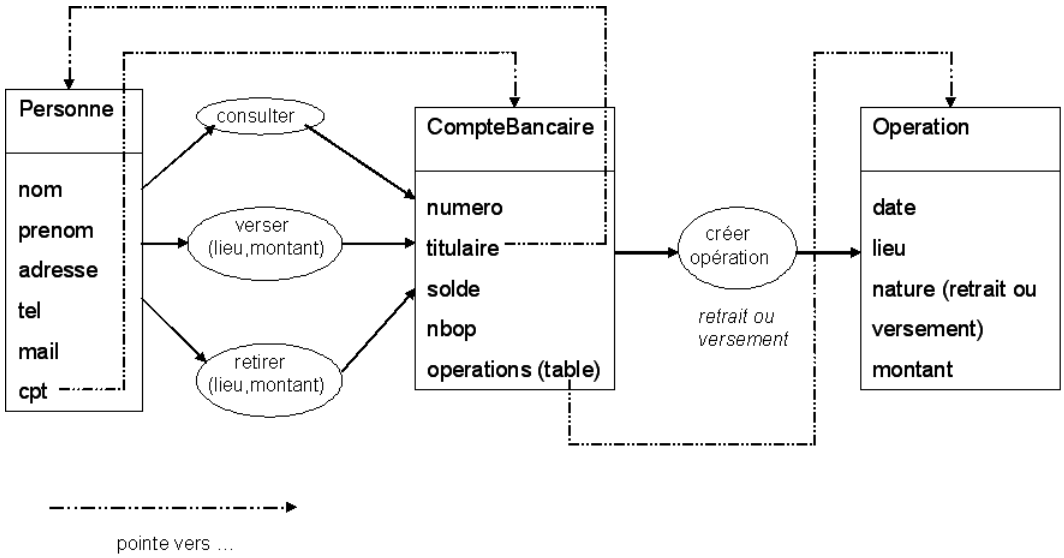
Une classe décrit une **structure de données** et un ensemble de **comportements**. Tous les objets provenant de la même classe ont la même **structure de données** et le même **comportement**.

La classe précise **qui** a accès à **quoi** en utilisant des **qualificatifs de protection** (private, protected, public). Pour permettre d'accéder à des attributs privés, elle doit donner des méthodes d'accès (lecture et / ou écriture) à ces attributs. Sinon ils seraient inaccessibles de l'extérieur de la classe.

Exemple

La classe **CompteBancaire** définit ce que sera un compte bancaire. Chaque CompteBancaire aura une structure (il contiendra un numéro, un titulaire, un ensemble opérations…) et un ensemble de comportements (prendre en compte un versement, un retrait, donner son solde…) puisqu'un client qui a un compte bancaire peut le consulter, y verser ou en retirer de l'argent.

Ce problème nécessite une classe **Personne**, une classe **CompteBancaire** et une classe **Operation**, chacune avec une **structure de données** propre et des **comportements** :



Classe Personne

```
class Personne {
        protected String nom ;                  // liste des attributs d'instance
        protected String prenom ;
        protected String adresse ;
        protected String tel ;
        protected String mail ;
        protected CompteBancaire cpt ;          // référence (adresse mémoire) d'un futur compte

        // méthode de création d'une personne (laquelle n'a pas encore de compte bancaire)
        Personne(String no, String pn, String adr, String tl, String em) {
                nom = no ;
                prenom = pn ;
                adresse = adr ;
                tel = tl ;
                mail = em ;
                // pas encore de valeur donnée encore à cpt, donc par défaut cpt = null
                }
        }
```

protected :
String :
`Personne(String no, String pn, String adr, String tl, String em) :` **…** de la classe Personne.

Constructeur :

Classe Operation

```
class Operation {
        protected Date date ;
        protected String lieu ;
        protected String nature ;
        private double montant ;

        Operation(Date d, String l, String n, double m) {
                date = d ;
                lieu = l ;
                nature = n ;
                montant = m ;
                }
        }
```

Date date :

Nécessite une librairie :

String nature :

`Operation(Date d, String l, String n, double m)` : **constructeur** de la classe Operation.

Classe CompeBancaire

```
class CompteBancaire {

	private int num ;
	private Personne titulaire ;
	private double solde ;
	private Operation operations[] ;
	private int nbop ;

	// méthode de création d'un CompteBancaire par une Personne p avec un montant initial m

	CompteBancaire(Personne p, double m) {
		num = 1 ;				// numéro à attribuer par la banque
		titulaire = p ;
		solde = m ;
		operations = new Operation[100] ;	// tableau pour les opérations futures
		nbop = 0 ;
		operations[nbop] = new Operation(new Date(), "agence", "versement", m);
		nbop++ ;
		p.cpt = this ;				// this : le compte juste créé
		}
	}
```

private :

double solde :

operations[] operations :

`CompteBancaire(Personne p, double m)` : **constructeur** de la classe CompteBancaire.

`p.cpt = this` :

Classe utilCompte1 : programme principal (phase 1)

```
public class UtilCompte1 {

	public static void main(String[] args) {

		Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
			"0144415665", "durand@orange.fr");
		CompteBancaire compte1 = new CompteBancaire(p1, 300.00);

		Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
			Paris", "0144400000", "louise.albert@orange.fr");
		CompteBancaire compte2 = new CompteBancaire(p2, 500.00);
		}
	}
```

`public class UtilCompte1` :

`void` :

`static main` : indique une **méthode de classe** que seule la classe peut effectuer.

`main(String[] args)` :

Fichier complet (version 1) : UtilCompte1.java

```
import java.util.* ;

class Personne {
	protected String nom ;		// liste des attributs
	protected String prenom ;
	protected String adresse ;
	protected String tel ;
	protected String mail ;
	protected CompteBancaire cpt ;	// référence (adresse mémoire) d'un futur compte

	// méthode de création d'une personne (laquelle n'a pas encore de compte bancaire)
	Personne(String no, String pn, String adr, String tl, String em) {
		nom = no ;
		prenom = pn ;
		adresse = adr ;
		tel = tl ;
		mail = em ;
		// pas de valeur donnée encore à cpt, donc par défaut cpt = null
		}
	}

class Operation {
	protected Date date ;
	protected String lieu ;
	protected String nature ;
	private double montant ;

	Operation(Date d, String l, String n, double m) {
		date = d ;
		lieu = l ;
		nature = n ;
		montant = m ;
		}
	}

class CompteBancaire {
	private int num ;
	private Personne titulaire ;
	private double solde ;
	private Operation operations[] ;
	private int nbop ;

	// méthode de création d'un CompteBancaire par une Personne p avec un montant m
	CompteBancaire(Personne p, double m) {
		num = 1 ;				// numéro à attribuer par la banque
		titulaire = p ;
		solde = m ;
		operations = new Operation[100] ;	// tableau pour les opérations futures
		nbop = 0 ;
		operations[nbop] = new Operation(new Date(), "agence", "versement", m);
		nbop++ ;
		p.cpt = this ;		// this : le compte juste créé
		}
	}

public class UtilCompte1 {

	public static void main(String[] args) {
		Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
			"0144415665", "durand@orange.fr");
		CompteBancaire compte1 = new CompteBancaire(p1, 300.00);

		Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
			Paris", "0144400000", "louise.albert@orange.fr");
		CompteBancaire compte2 = new CompteBancaire(p2, 500.00);
		}
	}
```

Une seule classe…
Application :

Le fichier doit s'appeler …

Compilation : **javac UtilCompte1.java** (appel au compilateur) → **Attention à la casse !!!**

→ Génération de quatre fichiers de pseudo-code (vérifier avec la commande **dir**).

Interprétation : **java UtilCompte1** (appel à l'interpréteur) crée les objets mais n'affiche RIEN.

Spécificité de java : compilation + exécution

Etape de compilation :

Etape d'interprétation :

Portabilité :

Les comportements : méthodes d'instance

Une Personne (instance de la classe Personne) doit pouvoir **consulter** son compte, y **verser** ou en **retirer** de l'argent. Elle doit donc disposer des opérations consulter, verser et retirer implémentées sous forme de **méthodes** `consulter(), verser(...), retirer(...)`.

Mais la Personne **ne peut pas modifier directement le solde** de son CompteBancaire car solde est déclaré **private** dans CompteBancaire. Il faut que la classe CompteBancaire prévoie les trois opérations correspondantes, implémentées sous forme de **méthodes** `donneSolde(), verse(...), retire(...).`

La **Personne** demande alors à son **CompteBancaire** de faire l'**Opération** attendue.

```
class Personne {
	protected String nom ;
	protected String prenom ;
	protected String adresse ;
	protected String tel ;
	protected String mail ;
	protected CompteBancaire cpt ;	// référence vers un futur compte

	// méthode de création d'une personne 'qui n'a pas encore de compte bancaire)
	Personne(String no, String pn, String adr, String tl, String em) {
		nom = no ;
		prenom = pn ;
		adresse = adr ;
		tel = tl ;
		mail = em ;
		// pas encore de cpt donc cpt = null
		}

	void consulter() {
		// une personne consulte son compte
		System.out.println("Titulaire : "+ nom + " " + prenom);
		System.out.println("Solde : "+cpt.donneSolde());
		System.out.println("-----------------------------------");
		}

	void verser(String lieu, double montant) {
		cpt.verse(lieu, montant) ;
		}

	void retirer(String lieu, double montant) {
		cpt.retire(lieu, montant) ;
		}
	}
```

`consulter, verser, retirer` : opérations (dites …) de la classe Personne

`System.out.println(...)` :

```
class CompteBancaire {
	private int num ;
	private Personne titulaire ;
	private double solde ;
	private Operation operations[] ;
	private int nbop ;

	// méthode de création d'un CompteBancaire par une Personne p avec un montant initial m
	CompteBancaire(Personne p, double m) {
		num = 1 ;		// numéro à attribuer par la banque
		titulaire = p ;
		solde = m ;
		operations = new Operation[100] ;	// tableau pour les opérations futures
		nbop = 0 ;
		operations[nbop] = new Operation(new Date(), "agence", "versement", m);
		nbop++ ;
		p.cpt = this ;		// this : le compte juste créé
		}

	double donneSolde() {
		// permet de lire le solde
		return solde ;
		}

	void verse(String lieu, double m) {
		// permet de verser le montant m sur le compte en enregistrant une opération
		operations[nbop] = new Operation(new Date(), lieu, "versement", m);
		nbop++ ;
		solde += m ;
		System.out.println("versement de " + m + " effectue");
		System.out.println("-----------------------------------");
		}

	void retire(String lieu, double m) {
		// permet de retirer le montant m du compte en enregistrant une opération
		if (solde >= m) {
			solde -= m ;
			operations[nbop] = new Operation(new Date(), lieu, "retrait", m);
			nbop++ ;
			System.out.println("retrait de " + m + " effectue");
			}
		else {
			System.out.println("retrait de " + m + " refuse");
			}
		System.out.println("-----------------------------------");
		}
	}
```

`double donneSolde()` : méthode avec valeur de …

Classe utilCompte2 : programme principal (phase 2)

On ajoute dans le programme principal la consultation du solde de comptes, un versement et un retrait :

```
public class UtilCompte2 {

	public static void main(String[] args) {
		Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
			"0144415665", "durand@orange.fr");
		CompteBancaire compte1 = new CompteBancaire(p1, 300.00); p1.consulter() ;

		Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
			Paris", "0144400000", "louise.albert@orange.fr");
		CompteBancaire compte2 = new CompteBancaire(p2, 500.00); p2.consulter() ;

		p1.verser("agence", 100.00) ; p1.consulter() ;

		p2.retirer("distributeur A", 200.00); p2.consulter() ;
		}
	}
```

Fichier complet (version 2) : UtilCompte2.java

```
import java.util.* ;

class Personne {
	protected String nom ;
	protected String prenom ;
	protected String adresse ;
	protected String tel ;
	protected String mail ;
	protected CompteBancaire cpt ;	// référence vers un futur compte

	// méthode de création d'une personne qui n'a pas encore de compte bancaire
	Personne(String no, String pn, String adr, String tl, String em) {
		nom = no ;
		prenom = pn ;
		adresse = adr ;
		tel = tl ;
		mail = em ;
		// pas encore de cpt donc cpt = null
		}

	void consulter() {
		// une personne consulte son compte
		System.out.println("Titulaire : "+ nom + " " + prenom);
		System.out.println("Solde : "+cpt.donneSolde());
		System.out.println("-----------------------------------");
		}

	void verser(String lieu, double montant) {
		cpt.verse(lieu, montant) ;
		}

	void retirer(String lieu, double montant) {
		cpt.retire(lieu, montant) ;
		}
	}

class Operation {
	protected Date date ;
	protected String lieu ;
	protected String nature ;
	private double montant ;

	Operation(Date d, String l, String n, double m) {
		date = d ;
		lieu = l ;
		nature = n ;
		montant = m ;
		}
	}

class CompteBancaire {
	private int num ;
	private Personne titulaire ;
	private double solde ;
	private Operation operations[] ;
	private int nbop ;

	CompteBancaire(Personne p, double m) {
		num = 1 ;		// numéro à attribuer par la banque
		titulaire = p ;
		solde = m ;
		operations = new Operation[100] ;	// tableau pour les opérations futures
		nbop = 0 ;
		operations[nbop] = new Operation(new Date(), "agence", "versement", m);
		nbop++ ;
		p.cpt = this ;		// this : le compte juste créé
		}

	double donneSolde() {
		// permet de lire le solde
		return solde ;
		}
```

```
        void verse(String lieu, double m) {
                // permet de verser le montant m sur le compte en enregistrant une opération
                operations[nbop] = new Operation(new Date(), lieu, "versement", m);
                nbop++ ;
                solde += m ;
                System.out.println("versement de " + m + " effectue");
                System.out.println("-----------------------------------");
                }

        void retire(String lieu, double m) {
                // permet de retirer le montant m du compte en enregistrant une opération
                if (solde >= m) {
                        solde -= m ;
                        operations[nbop] = new Operation(new Date(), lieu, "retrait", m);
                        nbop++ ;
                        System.out.println("retrait de " + m + " effectue");
                        }
                else {
                        System.out.println("retrait de " + m + " refuse");
                        }
                System.out.println("-----------------------------------");
                }
        }

public class UtilCompte2 {
        public static void main(String[] args) {
                Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
                        "0144415665", "durand@orange.fr");
                CompteBancaire compte1 = new CompteBancaire(p1, 300.00); p1.consulter() ;

                Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
                        Paris", "0144400000", "louise.albert@orange.fr");
                CompteBancaire compte2 = new CompteBancaire(p2, 500.00); p2.consulter() ;

                p1.verser("agence", 100.00) ; p1.consulter() ;

                p2.retirer("distributeur A", 200.00); p2.consulter() ;
                }
        }
```

Compilation : **javac UtilCompte2.java** (appel au compilateur) → **Attention à la casse !!!**

→ Génération de quatre fichiers de pseudo-code (vérifier avec la commande **dir**).

Interprétation : **java UtilCompte2** (appel à l'interpréteur). Notez ci-dessous les résultats affichés en face des instructions correspondantes :

Instruction	Affichage ou résultat
`Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris", "0144415665", "durand@orange.fr");`	*Création d'un objet en mémoire*
`CompteBancaire compte1 = new CompteBancaire(p1, 300.00);`	*Création d'un objet en mémoire*
`p1.consulter() ;`	
`Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005 Paris", "0144400000","louise.albert@orange.fr");`	*Création d'un objet en mémoire*
`CompteBancaire compte2 = new CompteBancaire(p2, 500.00);`	*Création d'un objet en mémoire*
`p2.consulter() ;`	
`p1.verser("agence", 100.00) ;`	
`p1.consulter() ;`	
`p2.retirer("distributeur A", 200.00);`	
`p2.consulter() ;`	

Les attributs et méthodes de classe

Dans la classe CompteBancaire, on souhaite disposer d'un attribut **nbcomptes** contenant le nombre de comptes effectivement ouverts. On le déclare précédé du mot **`static`** pour en faire un attribut dit **de classe** qui correspond à une variable globale pour la classe.

Attribut de classe : …

Pour faire afficher un attribut de classe, il faut définir dans la classe CompteBancaire une méthode spécifique, elle-même déclarée **`static`** : `static void afficheNombreComptes()`

Méthode de classe : …

```
class CompteBancaire {
       static int nbcomptes = 0 ;           // un attribut de classe pour compter les comptes
       private int num ;                    // les autres sont privés
       private Personne titulaire ;
       private double solde ;
       private Operation operations[] ;
       private int nbop ;

       // méthode de création d'un CompteBancaire par une Personne p avec un montant initial m
       CompteBancaire(Personne p, double m) {
              num = 1 ;              // numéro à attribuer par la banque
              titulaire = p ;
              solde = m ;
              operations = new Operation[100] ;    // tableau pour les opérations futures
              nbop = 0 ;
              operations[nbop] = new Operation(new Date(), "agence", "versement", m);
              nbop++ ;
              p.cpt = this ;         // this : le compte juste créé
              nbcomptes++ ;          // il y a un compte de plus : nbcomptes ← nbcomptes + 1
              }

       double donneSolde() {
              // permet de lire le solde
              return solde ;
              }

       void verse(String lieu, double m) {
              // permet de verser le montant m sur le compte en enregistrant une opération
              operations[nbop] = new Operation(new Date(), lieu, "versement", m);
              nbop++ ;
              solde += m ;
              System.out.println("versement de " + m + " effectue");
              System.out.println("-----------------------------------");
              }

       void retire(String lieu, double m) {
              // permet de retirer le montant m du compte en enregistrant une opération
              if (solde >= m) {
                     solde -= m ;
                     operations[nbop] = new Operation(new Date(), lieu, "retrait", m);
                     nbop++ ;
                     System.out.println("retrait de " + m + " effectue");
                     }
              else {
                     System.out.println("retrait de " + m + " refuse");
                     }
              System.out.println("-----------------------------------");
              }

       static void afficheNombreComptes() {
              System.out.println("Il y a " + nbcomptes + " compte(s).");
              System.out.println("-----------------------------------");
              }
       }
```

Programme principal (phase 3)

On veut faire afficher le nombre de comptes plusieurs fois dans le programme principal :

```
public class UtilCompte3 {
	public static void main(String[] args) {
		CompteBancaire.afficheNombreComptes() ;

		Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
			"0144415665", "durand@orange.fr");
		CompteBancaire compte1 = new CompteBancaire(p1, 300.00); p1.consulter() ;

		CompteBancaire.afficheNombreComptes() ;

		Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
			Paris", "0144400000", "louise.albert@orange.fr");
		CompteBancaire compte2 = new CompteBancaire(p2, 500.00); p2.consulter() ;

		CompteBancaire.afficheNombreComptes() ;

		p1.verser("agence", 100.00) ; p1.consulter() ;
		p2.retirer("distributeur A", 200.00); p2.consulter() ;
		}
	}
```

```
	CompteBancaire.afficheNombreComptes() :
```

Fichier complet : UtilCompte3.java : il comporte les quatre classes et doit porter le nom de la classe maître (celle qui contient la méthode main) :

UtilCompte3.java

```
import java.util.*;

class Personne {
	protected String nom ;			// liste des attributs
	protected String prenom ;
	protected String adresse ;
	protected String tel ;
	protected String mail ;
	protected CompteBancaire cpt ;		// référence (adresse mémoire) d'un futur compte

	// méthode de création d'une personne (laquelle n'a pas encore de compte bancaire)
	Personne(String no, String pn, String adr, String tl, String em) {
		nom = no ;
		prenom = pn ;
		adresse = adr ;
		tel = tl ;
		mail = em ;
		// pas de valeur donnée encore à cpt, donc par défaut cpt = null
		}

	void consulter() {
		// une personne consulte son compte
		System.out.println("Titulaire : "+ nom + " " + prenom);
		System.out.println("Solde : "+cpt.donneSolde());
		System.out.println("-----------------------------------");
		}

	void verser(String lieu, double montant) {
		cpt.verse(lieu, montant) ;
		}
	void retirer(String lieu, double montant) {
		cpt.retire(lieu, montant) ;
		}
	}

class Operation {
	protected Date date ;
	protected String lieu ;
	protected String nature ;
	private double montant ;
	Operation(Date d, String l, String n, double m) {
		date = d ;
		lieu = l ;
		nature = n ;
		montant = m ;
		}
	}
```

```
class CompteBancaire {
	static int nbcomptes = 0 ;		// un attribut de classe pour compter les comptes
	private int num ;			// les autres sont privés
	private Personne titulaire ;
	private double solde ;
	private Operation operations[] ;
	private int nbop ;

	// méthode de création d'un CompteBancaire par une Personne p avec un montant m
	CompteBancaire(Personne p, double m) {
		num = 1 ;		// numéro à attribuer par la banque
		titulaire = p ;
		solde = m ;
		operations = new Operation[100] ;	// tableau pour les opérations futures
		nbop = 0 ;
		operations[nbop] = new Operation(new Date(), "agence", "versement", m);
		nbop++ ;
		p.cpt = this ;		// this : le compte juste créé
		nbcomptes++ ;		// il y a un compte de plus : nbcomptes ← nbcomptes + 1
		}

	double donneSolde() {
		// permet de lire le solde
		return solde ;
		}

	void verse(String lieu, double m) {
		// permet de verser le montant m sur le compte en enregistrant une opération
		operations[nbop] = new Operation(new Date(), lieu, "versement", m);
		nbop++ ;
		solde += m ;
		System.out.println("versement de " + m + " effectue");
		System.out.println("-------------------------------------");
		}

	void retire(String lieu, double m) {
		// permet de retirer le montant m du compte en enregistrant une opération
		if (solde >= m) {
			solde -= m ;
			operations[nbop] = new Operation(new Date(), lieu, "retrait", m);
			nbop++ ;
			System.out.println("retrait de " + m + " effectue");
			}
		else { System.out.println("retrait de " + m + " refuse"); }
		System.out.println("-------------------------------------");
		}

	static void afficheNombreComptes() {
		System.out.println("Il y a " + nbcomptes + " compte(s).");
		System.out.println("-------------------------------------");
		}
	}

public class UtilCompte3 {
	public static void main(String[] args) {
		CompteBancaire.afficheNombreComptes() ;
		Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris",
			"0144415665", "durand@orange.fr");
		CompteBancaire compte1 = new CompteBancaire(p1, 300.00); p1.consulter() ;
		CompteBancaire.afficheNombreComptes() ;
		Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005
			Paris", "0144400000", "louise.albert@orange.fr");
		CompteBancaire compte2 = new CompteBancaire(p2, 500.00); p2.consulter() ;
		CompteBancaire.afficheNombreComptes() ;
		p1.verser("agence", 100.00) ; p1.consulter() ;
		p2.retirer("distributeur A", 200.00); p2.consulter() ;
		}
	}
```

Compilation : **javac UtilCompte3.java** (appel au compilateur) → **Attention à la casse !!!**

Interprétation : **java UtilCompte3** (appel à l'interpréteur) donne le résultat suivant :

Instruction	Affichage ou résultat
`CompteBancaire.afficheNombreComptes() ;`	
`Personne p1 = new Personne("Durand", "Etienne", "96 rue d'Assas, 75006 Paris","0144415665","durand@orange.fr");`	*Création d'un objet en mémoire*
`CompteBancaire compte1 = new CompteBancaire(p1, 300.00);`	*Création d'un objet en mémoire*
`p1.consulter() ;`	
`CompteBancaire.afficheNombreComptes() ;`	
`Personne p2 = new Personne("Albert", "Louise", "12 place du Panthéon, 75005 Paris", "0144400000","louise.albert@orange.fr");`	*Création d'un objet en mémoire*
`CompteBancaire compte2 = new CompteBancaire(p2, 500.00);`	*Création d'un objet en mémoire*
`p2.consulter() ;`	
`CompteBancaire.afficheNombreComptes() ;`	
`p1.verser("agence", 100.00) ;`	
`p1.consulter() ;`	
`p2.retirer("distributeur A", 200.00);`	
`p2.consulter() ;`	

Exercice

Dans le programme principal, créer une troisième personne, lui ouvrir un compte, effectuer plusieurs versements et retraits.
Enregistrer le fichier, le recompiler et l'exécuter pour vérifier son bon fonctionnement.

Attributs et méthodes : bilan

Une classe définit des **attributs** et des **comportements** (appelés des **méthodes**).

Une classe sert généralement à construire des objets (instances), mais ce n'est pas obligatoire.

→ Exemple : la classe principale UtilCompte ne construit pas d'objet.

Tout attribut précédé du mot clé **static** est un **attribut de classe**. Il appartient à la classe, mais pas aux objets générés (il n'existe donc qu'en un seul exemplaire en mémoire). Seule une méthode de classe peut modifier un attribut de classe.

→ Exemple : nbcomptes dans la classe CompteBancaire.

Tous les attributs non précédés du mot clé static sont des **attributs d'instance**. Ils sont **répliqués** dans chaque objet généré.

→ Exemples : nom, prenom, adresse, tel, mail, cpt dans la classe Personne ; num, titulaire, solde, operations, nbop dans la classe CompteBancaire ; date , lieu, nature, montant dans la classe Operation.

Toute méthode précédée de **static** est une **méthode de classe**. Elle ne peut être effectuée que par la classe.

→ Exemples : main dans UtilCompte, afficheNombreComptes dans CompteBancaire.

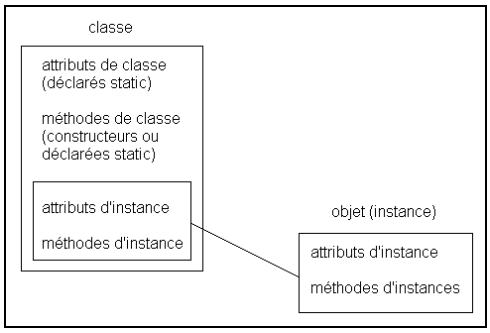
Un **constructeur** est une méthode de classe, même s'il n'est pas précédé de static. Il porte le nom de la classe et n'a pas de type de retour. C'est la seule méthode permettant de construire des objets.

→ Exemple : les trois constructeurs des classes Personne, CompteBancaire et Operation.

Toute méthode non précédée de static est une **méthode d'instance**. Elle ne peut être exécutée que par des **objets** (instances).

→ Exemples : toutes les méthodes, sauf le main, les trois constructeurs et afficheNombreComptes.

Schéma :



Tout objet généré comporte tous les attributs d'instance et peut effectuer chacune des méthodes d'instance.

Il peut lire les attributs de classe (mais pas les modifier).

Les structures de contrôle

Le test if … else

Il s'écrit :

```
if (test) {
        instructions;
        }
else    {
        instructions;
        }
```

La clause else est facultative.
On peut avoir simplement :

```
if (test) {
        instructions ;
        }
```

Exemple : tester dans la classe CompteBancaire s'il y a eu au moins une opération sur un CompteBancaire.

```
if (nbop > 0) {
       System.out.println("au moins une operation") ;
       }
```

Les boucles

La boucle for

La boucle for permet de répéter un certain nombre de fois des instructions. Elle s'écrit :
`for (initialisation; test_continuation; incrémentation) {instructions;}`

L'initialisation permet de préciser la variable qui contrôle la boucle et de l'initialiser. Exemple : int i=0

Le test de continuation dit quel est le test qui doit être vérifié pour que la boucle continue. Exemple : i < 20

L'incrémentation précise comment la variable doit évoluer entre deux itérations. Généralement on écrit i++ qui veut dire i=i+1 (on passe de i à i+1).

Exemple : lister dans la classe CompteBancaire les lieux où ont eu lieu les diverses opérations sur le Comptebancaire.

```
if (nbop > 0) {
       for (int i = 0; i < nbop; i++){
              System.out.println("Operation "+i+" : "+ operations[i].lieu);
              }
       }
```

`operations[i]` :

`operations[i].lieu` : dans Operation, lieu est déclaré **protégé** : il est directement accessible par les autres classes du même fichier, dont la classe CompteBancaire.

La boucle while

La boucle while permet de **répéter** des instructions tant qu'un **test** est vérifié.

Elle s'écrit :

```
while (test) {
   instructions ;
   }
```

Les instructions doivent modifier **au moins** une variable intervenant dans le test, pour que le while ne boucle pas indéfiniment.

Exemple : lister dans la classe CompteBancaire les lieux où ont eu lieu les diverses opérations sur le Comptebancaire.

```
int i = 0 ;
while (i < nbop) {
	System.out.println("Operation "+i+" : "+ operations[i].lieu);
	i++ ;
	}
```

La boucle do … while

Elle permet de faire exécuter au moins une fois des instructions puis de les répéter tant qu'un test est vérifié.

Elle s'écrit :

```
do	{
	instructions ;
	}
while (test);
```

Ces instructions doivent modifier au moins une variable du test, pour que le do … while ne boucle pas indéfiniment.

Exemple : lister dans la classe CompteBancaire les lieux où ont eu lieu les opérations sur le CompteBancaire.

```
if (nbop > 0) {
	i = 0 ;
	do	{
		System.out.println("Operation "+i+" : "+ operations[i].lieu);
		i++ ;
		}
	while (i < nbop) ;
	}
```

Les sauts

Le break

L'instruction **break** permet de terminer un test, une boucle ou un bloc d'instructions en reprenant l'exécution directement après le test, la boucle ou le bloc (on saute donc tous les cas suivants).

Exemple : lister dans la classe CompteBancaire les lieux où ont eu lieu les diverses opérations sur le Comptebancaire.

```
for (int i = 0; i < 100 ; i++){
	if (i == nbop) {
		break ;
		}
	else {
		System.out.println("Operation "+i+" : "+ operations[i].lieu);
		}
	}
```

`==` veut dire …

Le continue

L'instruction **continue** permet de sortir d'un pas d'itération en reprenant l'exécution à l'itération suivante. On arrête donc les instructions de l'itération en cours pour passer à la suivante.

Exemple : lister dans la classe CompteBancaire les lieux où ont eu lieu les diverses opérations sur le Comptebancaire, sauf la première (ouverture du compte).

```
if (nbop > 0) {
	for (int i = 0; i < nbop; i++){
		if (i==0) continue ;
		System.out.println("Operation "+i+" : "+ operations[i].lieu);
		}
	}
```

Le return

L'instruction **return** permet de sortir d'une méthode sans nécessairement attendre la fin de celle-ci. Cette instruction ne peut donc pas figurer directement dans la méthode main.

Exemple : dans une méthode, chercher le premier lieu différent de l'agence où a lieu une opération :

```
if (nbop > 0) {
       for (int i = 0; i < nbop; i++){
              if(!operations[i].lieu.equalsIgnoreCase("agence"))
                     return operations[i].lieu ;
              }
       }
```

`!` veut dire …

`equalsIgnoreCase` :

Exercice

Compléter le fichier UtilCompte.java de telle sorte qu'une Personne dispose de méthodes :

- lui permettant de lister les lieux et les natures (retrait ou versement) de ses opérations. Exemple :
  ```
  Operation 0 : versement lieu : agence
  Operation 1 : retrait lieu : distributeur A
  ```

- lui permettant de lister l'ensemble de ses opérations. Exemple :
  ```
  Operation 0 : versement de 500.0 lieu : agence
  Operation 1 : retrait de 200.0 lieu : distributeur A
  ```

- lui permettant de lister ses opérations d'un montant supérieur à un montant m donné.

Tester ces méthodes en complétant le programme principal.

Héritage et polymorphisme

Héritage

Une classe peut **dériver** d'une autre classe :

```
class fille extends mere {
    ...
    }
```

→ cela établit une **relation hiérarchique** entre classes.

La classe ancêtre de toutes les classes est la classe **Object**. Si aucune relation d'héritage n'est précisée dans la définition d'une classe, la classe dérive automatiquement d'Object.

Toute classe (sauf Object) **hérite** des propriétés (attributs, méthodes) des classes supérieures (appelées super-classes) et y ajoute ses propres attributs et méthodes ; d'où l'usage du mot **extends** pour étendre la classe mère. Une classe peut ainsi être définie approximativement et être affinée ensuite dans des sous-classes plus précises.

Exemple :
Une classe `CompteBancaire` peut avoir comme sous-classes les classes `CompteATerme` et `CompteLivret`.
Les sous-classes auront chacune un attribut taux que la classe `CompteBancaire` n'a pas et la classe `CompteATerme` aura une gestion du temps que la classe `CompteLivret` n'aura pas.

Polymorphisme

Le **polymorphisme** découle de l'héritage : une même méthode peut se comporter différemment selon la classe dans laquelle elle est exécutée.

C'est un mécanisme qui permet à une sous-classe de **redéfinir** une méthode dont elle a hérité tout en lui conservant **la même signature** (même nom, mêmes arguments, même valeur de retour).

On peut alors avoir, dans une classe mère et ses sous-classes, une méthode de même signature avec des codes différents. L'appel à la méthode déclenchera des traitements différents selon la classe de l'objet qui l'exécute.

Exemple :
Un `CompteBancaire` peut avoir une méthode `void affiche()` qui affiche le titulaire et le solde. Un `CompteLivret` pourra avoir une méthode `void affiche()` qui affiche en plus un taux, et un `CompteATerme` aura lui aussi une méthode `void affiche()`qui affichera en outre les dates du placement.

La méthode `void affiche()` a toujours la même **signature**, avec des **corps différents** → polymorphisme.

Si on a une instruction `cpt.affiche()`, alors :

- si `cpt` est une instance de `CompteBancaire`, la première méthode s'exécutera
- si `cpt` est une instance de `CompteLivret`, la deuxième méthode s'exécutera
- si `cpt` est une instance de `CompteATerme`, la troisième méthode s'exécutera.

Les paquetages

Les paquetages (packages) sont des librairies de **classes,** d'**interfaces** et de **classes d'exceptions** regroupées selon leur fonction. Ils sont fournis en même temps que le compilateur javac et l'interpréteur java.

Principaux paquetages : java.lang, java.util, java.awt, java.awt.event.

- java.lang : classes fournissant les éléments de base du langage
- java.util : classes utilitaires
- java.awt : classes graphiques
- java.awt.event : classes gérant les événements sur les composants graphiques

La documentation concernant chaque paquetage est disponible sous forme de pages HTML (ici, dans le dossier **C:\sun\appserver\docs\doc\docs\.**

Choisir la page **index.html** puis cliquer sur le lien intitulé : **Java 2 Platform … API Specification**).

Lorsqu'on sélectionne un paquetage, on obtient la liste des **classes**, des **interfaces** et des **classes d'exceptions** qui composent ce paquetage.

Les classes

Si l'on choisit une **classe** dans la liste des classes d'un paquetage, on obtient ses relations d'héritage ainsi que son descriptif (voir l'exemple de la classe `Double` **page 21**).

Lorsque l'on clique sur un lien, on obtient des détails sur l'élément concerné (dans l'ordre, attribut de classe, constructeur, méthode de classe ou d'instance).

Les interfaces

Une **interface** est une sorte de classe spéciale, dont toutes les méthodes sont sans code. En fait, une interface **propose des services**.

Exemple : `WindowListener` va surveiller ce qui se passe sur la fenêtre d'une application graphique.

Si on choisit une **interface** dans la liste des interfaces d'un paquetage, on obtient ses relations d'héritage ainsi que son descriptif (voir les interfaces `WindowListener` et `ActionListener` **page 22**).

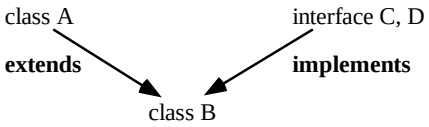
Une classe qui souhaite exploiter les services d'une interface le signale par le mot **implements** :

```
class B implements C {...} ;
```

On dit alors qu'elle **implémente** l'interface. Dans la classe B, on doit alors donner le code de **toutes** les méthodes de l'interface.

Une classe B ne peut hériter que d'une seule classe A, mais elle peut implémenter une ou plusieurs interfaces :

```
class B extends A implements C, D {...} ;
```



Les méthodes des interfaces C et D doivent alors **obligatoirement** être définies dans la classe B.

java.lang

Class Double — nom de la classe

java.lang.Object
└ java.lang.Number — héritage : Double dérive de Number, qui dérive de Object
└ java.lang.Double

All Implemented Interfaces:
Serializable, Comparable<Double>

```
public final class Double
extends Number
implements Comparable<Double>
```

final : on ne peut pas créer de sous-classe de cette classe
c'est donc une classe terminale

The `Double` class wraps a value of the primitive type `double` in an object. An object of type `Double` contains a single field whose type is `double`.

In addition, this class provides several methods for converting a `double` to a `String` and a `String` to a `double`, as well as other constants and methods useful when dealing with a `double`.

explication du rôle de la classe

Since:
JDK1.0
See Also:
Serialized Form

Field Summary — attributs de classe

static int	MAX_EXPONENT — nom de l'attribut Maximum exponent a finite `double` variable may have. — explication
static double	MAX_VALUE — nom de l'attribut A constant holding the largest positive finite value of type `double`, $(2-2^{-52})\cdot 2^{1023}$ — explication

Constructor Summary — constructeurs

Double(double value) Constructs a newly allocated `Double` object that represents the primitive `double` argument.	deux constructeurs : les paramètres doivent alors être différents (nombre et / ou nature)
Double(String s) Constructs a newly allocated `Double` object that represents the floating-point value of type `double` represented by the string.	

Method Summary — autres méthodes

byte	byteValue() — méthode d'instance ramenant un byte (valeur de retour) Returns the value of this `Double` as a `byte` (by casting to a `byte`).
static int	compare(double d1, double d2) — méthode de classe ramenant un int (valeur de retour) Compares the two specified `double` values.
int	compareTo(Double anotherDouble) — méthode d'instance ramenant un int (valeur de retour) Compares two `Double` objects numerically.
static long	doubleToLongBits(double value) — méthode de classe ramenant un long (valeur de retour) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout.
static long	doubleToRawLongBits(double value) — méthode de classe ramenant un long (valeur de retour) Returns a representation of the specified floating-point value according to the IEEE 754 floating-point "double format" bit layout, pre Not-a-Number (NaN) values.
double	doubleValue() — méthode d'instance ramenant un double (valeur de retour) Returns the `double` value of this `Double` object.
boolean	equals(Object obj) — méthode d'instance ramenant un boolean (valeur de retour) Compares this object against the specified object.

Descriptif de la classe Double du paquetage java.lang

java.awt.event

Interface WindowListener

All Superinterfaces:
EventListener

All Known Implementing Classes:
AWTEventMulticaster, BasicToolBarUI.FrameListener, JMenu.WinListener, WindowAdapter

---

```
public interface WindowListener
extends EventListener
```

déclaration de l'interface qui dérive de l'interface EventListener

The listener interface for receiving window events. The class that is interested in processing a window event either implements this interface (and all the methods it contains) or extends the abstract `WindowAdapter` class (overriding only the methods of interest). The listener object created from that class is then registered with a Window using the window's `addWindowListener` method. When the window's status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the `WindowEvent` is passed to it.

Since:
1.1
See Also:
`WindowAdapter`, `WindowEvent`, Tutorial: How to Write Window Listeners

Method Summary

void	`windowActivated(WindowEvent e)` Invoked when the Window is set to be the active Window.
void	`windowClosed(WindowEvent e)` Invoked when a window has been closed as the result of calling dispose on the window.
void	`windowClosing(WindowEvent e)` Invoked when the user attempts to close the window from the window's system menu.
void	`windowDeactivated(WindowEvent e)` Invoked when a Window is no longer the active Window.
void	`windowDeiconified(WindowEvent e)` Invoked when a window is changed from a minimized to a normal state.
void	`windowIconified(WindowEvent e)` Invoked when a window is changed from a normal to a minimized state.
void	`windowOpened(WindowEvent e)` Invoked the first time a window is made visible.

sept méthodes qui seront obligatoires dans toute classe qui voudra utiliser les services de l'interface il faudra alors donner le code de ces sept méthodes

Descriptif de l'interface WindowListener du paquetage java.awt.event

java.awt.event

Interface ActionListener

All Superinterfaces:
EventListener

All Known Subinterfaces:
Action

All Known Implementing Classes:
AbstractAction, AWTEventMulticaster, BasicDesktopPaneUI.CloseAction, BasicDesktopPaneUI.MaximizeAction, BasicDesktopPaneUI.MinimizeAction, BasicDesktopPaneUI.NavigateAction, BasicDesktopPaneUI.OpenAction,

liste partielle

```
public interface ActionListener
extends EventListener
```

The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

Since:
1.1
See Also:
`ActionEvent`, Tutorial: Java 1.1 Event Model

Method Summary

void	`actionPerformed(ActionEvent e)` Invoked when an action occurs.

Descriptif de l'interface ActionListener du paquetage java.awt.event

Les classes d'exception

Une **exception** est un objet, instance d'une des sous-classes de la classe **Exception** traitant des erreurs. Lorsqu'une méthode est exécutée, il peut se produire des erreurs. On dit alors qu'une Exception est **levée**.

Si on choisit une **classe d'exception** dans la liste des classes d'exception d'un paquetage, on obtient ses relations d'héritage ainsi que son descriptif (voir l'exemple de la classe `NumberFormatException` ci-dessous).

java.lang

Class NumberFormatException

java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
      └ java.lang.RuntimeException
        └ java.lang.IllegalArgumentException
          └ **java.lang.NumberFormatException**

All Implemented Interfaces:
Serializable

```
public class NumberFormatException
extends IllegalArgumentException
```

Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

Since:
JDK1.0

See Also:
`Integer.toString()`, Serialized Form

Constructor Summary

`NumberFormatException()`
Constructs a `NumberFormatException` with no detail message.

`NumberFormatException(String s)`
Constructs a `NumberFormatException` with the specified detail message.

Method Summary

Methods inherited from class java.lang.Throwable

`fillInStackTrace`, `getCause`, `getLocalizedMessage`, `getMessage`, `getStackTrace`, `initCause`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `setStackTrace`, `toString`

Descriptif de la classeNumberFormatException du paquetage java.lang

Chaque méthode qui risque de provoquer une ou des erreur(s) le signale par le mot **throws** suivi de la nature des erreurs susceptibles de se produire. Lorsqu'on a besoin d'utiliser une telle méthode risquant de provoquer des erreurs :

- on englobe l'appel à cette méthode dans un bloc **try** (pour capturer l'erreur)
- on précise ce qu'on fera en cas d'erreur dans une clause **catch** (pour traiter l'erreur).

Exemple : dans la classe FilterInputStream, une méthode de lecture d'un fichier est déclarée sous la forme : `public int read() throws IOException`. Le mot clé `throws` signale que cette méthode de lecture peut provoquer une erreur `IOException`, erreur d'entrée-sortie (si le fichier est manquant par exemple).
Pour l'utiliser, on écrira :

```
        try     {
                fichier.read() ;
                }
        catch (IOException) {
                System.out.println(" erreur de lecture ") ;
                return ;
                }
```

Le paquetage java.lang

java.lang = java.languages

Le paquetage **java.lang** est le **seul** paquetage dont l'emploi ne doit **jamais** être déclaré.

Il comprend notamment la classe **Object** à l'origine de toutes les classes, la classe **System**, les **classes enveloppantes**, la classe **Math** et la classe **String** (classe des chaînes de caractères).

La classe **System** contient des variables et des méthodes du système. Quand on écrit `System.out.println(...)`, on fait appel à l'attribut de classe **out** de la classe System.

Les variables des types de données primitifs (int, double, boolean, etc.) sont les **seuls éléments** qui ne sont pas des objets en java.

Mais on peut créer des objets à partir de ces types primitifs à l'aide des **classes enveloppantes** (Integer, Double, Boolean, etc.).

Exemples :

```
int n = 5 ;                                 // n n'est pas un objet
Integer obj1 = new Integer(n) ;             // obj1 est un objet
double f = 0.33 ;                           // f n'est pas un objet
Double obj2 = new Double(f) ;               // obj2 est un objet
boolean fini = false ;                      // fini n'est pas un objet
Boolean termine = new Boolean(fini) ;       // termine est un objet
```

Dans certains cas, on ne peut en effet sauvegarder que des objets. Le passage par les classes enveloppantes est alors impératif.

La classe **Math** contient toutes les méthodes nécessaires pour effectuer tous les calculs mathématiques. Ce sont toutes des **méthodes de classe**. On effectue donc l'appel à une méthode par une instruction : `Math.méthode(arguments)`

Exemples :

```
double j = -3.145 ;                         double k = Math.abs(j) ;
double l = Math.pow(j,k) ;                  double m = Math.sqrt(l) ;
```

La classe **String** contient un grand nombre de constructeurs permettant de construire des **chaînes de caractères** à partir d'entiers, de double, etc. et de très nombreuses méthodes pour le **traitement des chaînes de caractères**

Exercice

Consulter la page de documentation concernant la classe **String** de **java.lang** puis répondez aux questions suivantes :

- De combien de constructeurs cette classe dispose-t-elle ?
- Quelle est la différence entre eux ?
- Quelle est la méthode permettant de trouver la longueur d'une chaîne ?

Si s est une chaîne de caractères,

- qu'écrit-on pour récupérer dans un entier lg la longueur de s ?
- qu'écrit-on pour tester si s est égale à une autre chaîne de caractères t ?
- si s="Aujourd'hui" et t=" nous sommes lundi", comment obtient-on "Aujourd'hui nous sommes lundi" ?

Les autres paquetages

Le paquetage **java.util** contient des classes utilitaires, notamment **Calendar** et **GregorianCalendar** pour les dates, **Random** : pour les nombres aléatoires.

Les paquetages **java.awt** et **java.awt.event** permettent de créer des composants graphiques et de gérer les événements sur ces composants.

Le paquetage **java.io** comporte toutes les classes permettant de gérer les entrées-sorties.

Pour utiliser une partie d'un paquetage **autre** que java.lang, on utilise le mot-clé **import** :

Exemples

```
import java.util.GregorianCalendar ;
```
→ on se donne le droit d'utiliser la classe GregorianCalendar qui permet de créer des objets de type date

```
import java.util.Random ;
```
→ on se donne le droit d'utiliser la classe Random

```
import java.util.* ;
```
→ on se donne le droit d'utiliser **toutes** les classes de java.util.

Exercice

Consulter la page de documentation concernant la classe **Random** de java.util puis répondez aux questions suivantes :

- De combien de constructeurs cette classe dispose-t-elle ?
- Quelle est la différence entre eux ?

Si r est un objet de la classe Random (créé par l'instruction `r = new Random()` par exemple)

- qu'écrit-on pour récupérer dans un entier n un entier aléatoire compris entre 0 et 100 ?
- qu'écrit-on pour récupérer dans un flottant f un nombre réel flottant compris entre 0 et 1 ?

Annexes

Modes de protection des attributs

Les attributs d'une classe (qu'ils soient d'instance ou de classe) peuvent être déclarés **private**, **public**, **protected** ou ne pas être précédés de déclaration de protection.

Exemple :

```
class Truc {
        private int a ;
        protected int b ;
        public int c ;
        int d ;
        }
```

Déclaration private

Les attributs et méthodes déclarés **private** ne sont accessibles qu'à l'intérieur de leur propre classe

→ a ne sera donc accessible qu'aux méthodes de la classe Truc

Exemple : le solde d'un compte dans CompteBancaire, qui ne peut être lu par aucune autre classe. Pour y accéder, il faut obligatoirement passer par une des méthodes définies dans la classe CompteBancaire.

Déclaration protected

Les attributs et méthodes déclarés **protected** ne sont accessibles qu'aux sous-classes (qu'elles soient dans le même paquetage ou pas) et aux classes du même paquetage

→ b sera donc accessible aux sous-classes de Truc et aux classes du même paquetage que Truc

Exemple : les attributs d'une Personne qui doivent pouvoir être lus à partir d'un CompteBancaire, afin de retrouver le nom du titulaire du compte.

Déclaration public

Les attributs et méthodes déclarés **public** sont toujours accessibles

→ c sera accessible de toute classe

A éviter, car tout le monde peut accéder à ces attributs et modifier leur contenu.

Sans déclaration

Les attributs et méthodes **sans aucune déclaration de protection** sont accessibles par toutes les classes du même paquetage. C'est le mode de protection par défaut

→ d sera donc accessible par toutes les classes du même paquetage que Truc.

C'est un mode de protection un peu plus strict que le mode protected, puisque les sous-classes situées dans d'autres paquetages ne peuvent plus accéder à ces attributs.

Pratiquement, on utilise la plupart du temps le mode protected qui offre une protection correcte.

Mots réservés du langage

Les mots réservés (qui ne peuvent donc pas être employés comme noms de variables, de classes ou d'attributs sont les suivants.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	extends	false	final
finally	float	for	goto	if	implements
import	instanceof	int	interface	long	native
new	null	package	private	protected	public
return	short	static	strictfp	super	switch
synchronized	this	throw	throws	transient	true
try	void	volatile	while		