Développeur Web

Java

Date

Licence : CC BY-SA

SOMMAIRE

1 Classe et Objet

2 Héritage, classe abstraite, interface

- Héritage
- Classes et méthode abstraites
- Interface

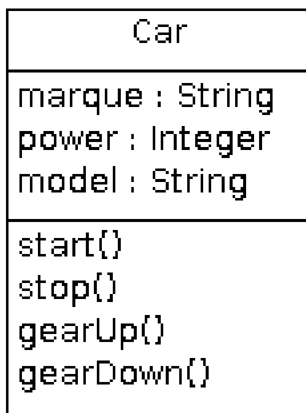
3 Packages

- Développement
- Industrialisation

Classe & Objet

Qu'est-ce qu'une classe ?

- Décrit une famille d'objets / Moule à fabriquer des objets
- Regroupe l'ensemble des caractéristiques commune de ces objets
  - Attributs, contenant les valeurs propres de chaque objet
  - Opérations caractérisant les services proposés par les objets de la classe



Modèle ISIS-Fish

<<entity>>
FisheryRegion

name : String
minLongitude : float
maxLongitude : float
minLatitude : float
maxLatitude : float
cellLengthLongitude : float
cellLengthLatitude : float
mapFiles : String
comment : String

getMapFileList() : List<String>
setMapFileList(v : List<String>) : void
getMapFilePath() : List<String>
getGear() : List<Gear>
getCell() : List<Cell>
getTripType() : List<TripType>
getVesselType() : List<VesselType>
getSpecies() : List<Species>
getZone() : List<Zone>
getStrategy() : List<Strategy>
getPort() : List<Port>
getMetier() : List<Metier>
getSetOfVessels() : List<SetOfVessels>
getObservations() : List<Observation>

<<entity>>
TripType

name : String
tripDuration : TimeUnit
minTimeBetweenTrip : TimeUnit
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
PopulationGroup

id : int
age : double
minLength : double
maxLength : double
comment : String

getLength() : double
getNaturalDeathRate(zone : Zone) : double
getMeanWeight() : double
getPrice() : double
isInLength(length : double) : boolean
getMaturityOgive() : double
getReproductionRate() : double
getFishingMortalityOtherFleets(zone : Zone) : double

<<entity>>
Population

name : String
geographicId : String
plusGroup : boolean
growth : Equation
growthReverse : Equation
comment : String
meanWeight : Equation
price : Equation
naturalDeathRate : Equation
fishingMortalityOtherFleets : Equation
caractBioComment : String
reproductionEquation : Equation
monthGapBetweenReproRecrutement : int
recruitmentDistribution : MatrixND
recruitmentEquation : Equation
recruitmentComment : String
zonesComment : String
seasonsComment : String
capturabilityComment : String
migrationComment : String
mappingZoneReproZoneRecru : MatrixND
capturability : MatrixND
capturabilityEquationUsed : boolean
capturabilityEquation : Equation
maturityOgiveEquation : Equation
reproductionRateEquation : Equation
fbarGroupMin : int
fbarGroupMax : int
abundanceReferenceMonth : Month
computeFOnLandings : boolean

getPopulationSeasonInfo(month : Month) : PopulationSeasonInfo
N2DToN1D(m : MatrixND) : MatrixND
split2D(m : MatrixND) : MatrixND
getNaturalDeathBirth(zone : Zone) : double
getAge(length : double,group : PopulationGroup) : double
getLength(age : double,group : PopulationGroup) : double
getNaturalDeathRateMatrix() : MatrixND
getRecruitmentMatrix(step : TimeStep,pop : Population,recruitmentInputs : RecruitmentInputMap,result : MatrixND) : MatrixND
getFishingMortalityOtherFleetsMatrix() : MatrixND

<<entity>>
PopulationSeasonInfo

reproductionDistribution : MatrixND
groupChange : boolean
simpleLengthChangeMatrix : boolean
lengthChangeMatrix : MatrixND
reproduction : boolean
useEquationMigration : boolean
migrationMatrix : MatrixND
emigrationMatrix : MatrixND
immigrationMatrix : MatrixND
migrationEquation : Equation
emigrationEquation : Equation
immigrationEquation : Equation

getCapturability(group : PopulationGroup) : double
spacializeLengthChangeMatrix(mat : MatrixND) : MatrixND
computeLengthChangeMatrix() : MatrixND
getGroupChangeMatrix(month : Month) : MatrixND
getMigrationMatrix(month : Month,N : MatrixND) : MatrixND
getEmigrationMatrix(month : Month,N : MatrixND) : MatrixND
getImmigrationMatrix(month : Month,N : MatrixND) : MatrixND
getReproductionMatrix(month : Month,N : MatrixND) : MatrixND
unspacializeLengthChangeMatrix(mat : MatrixND) : MatrixND

<<entity>>
VesselType

name : String
length : int
speed : double
maxTripDuration : TimeUnit
activityRange : double
minCrewSize : int
unitFuelCostOfTravel : double
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
Gear

name : String
effortUnit : String
standardisationFactor : double
parameterName : String
cost : double
possibleValue : RangeOfValues
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
StrategyMonthInfo

month : Month
minInactivityDays : double
numberOfTrips : double
proportionMetier : MatrixND

getFisheryRegion() : FisheryRegion
getProportionMetier(metier : Metier) : double
setProportionMetier(metier : Metier,prop : double) : void
getMetierWithProportion() : List<Metier>

<<entity>>
SetOfVessels

name : String
numberOfVessels : int
fixedCosts : double
vesselCosts : double
capitalDeprecation : double
interestCost : double
comment : String
technicalEfficiencyEquation : Equation

getFisheryRegion() : FisheryRegion
getTechnicalEfficiency(metier : Metier) : double

<<entity>>
Metier

name : String
gearParameterValue : String
comment : String
capturableSpeciesComment : String

getFisheryRegion() : FisheryRegion
getMetierSeasonInfo(month : Month) : MetierSeasonInfo
getGearParameterValueAsDouble() : double

<<entity>>
MetierSeasonInfo

seasonZoneComment : String
comment : String

getTargetFactor(group : PopulationGroup) : double
getCells() : List<Cell>

<<entity>>
Species

name : String
scientificName : String
codeRubbin : String
codeCEE : int
ageGroupType : boolean
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
Strategy

name : String
proportionSetOfVessels : double
comment : String
inactivityEquationUsed : boolean
inactivityEquation : Equation

getFisheryRegion() : FisheryRegion
getStrategyMonthInfo(month : Month) : StrategyMonthInfo
getInactivityDays(month : Month) : double
getProportionMetier() : MatrixND
setProportionMetier(strategyProportionMetier : MatrixND) : void

<<entity>>
Port

name : String
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
Cell

name : String
latitude : float
longitude : float
land : boolean
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
Zone

name : String
comment : String

getFisheryRegion() : FisheryRegion

<<entity>>
Season

getMonths() : List<fr.ifremer.isisfish.types.Month>
setMonths(listMonth : List<fr.ifremer.isisfish.types.Month>) : void
containsMonth(month : Month) : boolean

Construction d’une classe

- Le mot réservé « class » permet de construire une classe
- Chaque attribut (champs) doit être qualifié :
  - private : non accessible en dehors de la classe
  - public : accessible en dehors de la classe

Méthodes

- Définir des opérations sur les objets

```
visibility returnType methodName(<args>) { … }
```

- Visibility : public ou private
- returnType : type du retour attendu après l’appel de la méthode
- <args> : liste des arguments, sous format (type1 name1, type2 name2)

```
public int add(int x, int y) {
    return x+y ;
}
public void sayHello(String name) {
    System.out.println("Salut " + name);
}
```

Exemple

```java
public class Car {

    private String color;
    private String brand;
    private int power;

    public void modifer(String newColor, String newBrand, int newPower) {
        this.color = newColor;
        this.brand = newBrand;
        this.power = newPower;
    }

    public String getColor() {
        return color;
    }

    public void print() {
        System.out.println("Great " + color + " " + brand + ", " + power + " cv");
    }
}
```

Instanciation d'une classe

- Une classe est un « moule » à fabriquer des objets
- La création d'une instance se fait par l'opérateur « new »

```
public class Demo {

    public static void main(String[] args) {

        Car my106 = new Car();
        my106.modifier("green", "peugeot", 5);
        my106.print();
    }
}
```

- Les méthodes s'invoquent sur une instance de la classe, à l'aide de l'opérateur '.'

Polymorphisme

- Java permet de surcharger les méthodes

```
public class Car {

    private String color;
    private String brand;
    private int power;

    public void modifer(String newColor, String newBrand, int newPower) {
        this.color = newColor;
        this.brand = newBrand;
        this.power = newPower;
    }

    public void modifer(Car car) {
        this.color = car.color;
        this.brand = car.brand;
        this.power = car.power;
    }

    public String getColor() {
        return color;
    }

    public void print() {
        System.out.println("Great " + color + " " + brand + ", " + power + " cv");
    }
}
```

Les accesseurs

- Un objet n’est manipulable que par les méthodes de sa classe
- Toute manipulation extérieure d’une partie « private » entraîne une erreur de compilation
- Pour manipuler ces données, on utilise des méthodes spécifiques : les accesseurs
- Lecture ou écriture : ils doivent contrôler l’intégrité des données

Exemple

```java
public class Car {

    private String color;
    private String brand;
    private int power;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getPower() {
        return power;
    }

    public void setPower(int power) {
        this.power = power;
    }
    ...
}
```

```java
public class Demo {

    public static void main(String[] args) {

        Car my106 = new Car();
        my106.setColor("green");
        my106.setBrand("Peugeot");
        my106.setPower(5);
        my106.print();
    }
}
```

Initialiser les valeurs d'une instance

- Une instance comporte ses propres valeurs d'une classe : il est nécessaire des les initialiser

```
public class Car {

    private String color;
    private String brand;
    private int power;

    public void init(String color, String brand, int power) {
        this.color = color;
        this.brand = brand;
        this.power = power;
    }
  ...
}

public class Demo {

    public statci void main(String[] arg) {
        Car my106 = new Car();
        my106.init("green", "Peugeot", 5);
        my106.print();
    }
}
```

Le constructeur

- Une méthode d'initialisation n'est pas la solution la plus pertinente : cela nécessite un appel de la méthode tout de suite après la création de l'instance.
- Le « constructeur » résout cette problématique
- Le constructeur est une méthode spécifique :
  - Appelée automatiquement à chaque création d'instance
  - Homographe de la classe
  - Sans type de retour
  - Demandant à renseigner à la création de l'instance les arguments s'il en contient

Exemple

```
public class Car {

    private String color;
    private String brand;
    private int power;

    public Car(String color, String brand, int power) {
        this.color = color;
        this.brand = brand;
        this.power = power;
    }
    ...
}


public class Demo {

    public static void main(String[] args) {

        Car my106 = new Car("green", "Peugeot", 5);
        my106.print();
    }
}
```

Constructeurs multiples et par défaut

- Plusieurs constructeurs sont possibles avec différents paramètres

```java
public class Car {

    private String color;
    private String brand;
    private int power;

    public Car(String color) {
        this.color = color;
    }

    public Car(String color, String brand, int power) {
        this.color = color;
        this.brand = brand;
        this.power = power;
    }

    public Car(Car aCar) {
        this.color = aCar.getColor();
        this.brand = aCar.getBrand();
        this.power = aCar.getPower();
    }
    ...
}
```

Constructeurs multiples et par défaut

- Lorsqu'aucun constructeur n'est défini, c'est un constructeur sans paramètre ni instruction qui est créé.
- Ce constructeur par défaut peut également être précisé pour un traitement particulier

```java
public class Car {

    private String color;
    private String brand;
    private int power;

    public Car() {
        this.color = "gray";
        this.brand = "Kia";
        this.power = 7;
    }
    ...
}
```

Initialisation des attributs

- Il est possible d’initialiser directement des valeurs pour les attributs d’une classe
- Dans ce cas, l’initialisation par un constructeur par défaut n’a plus de sens

```
public class Car {

    private String color = "gray";
    private String brand = "Kia";
    private int power = 7;

    public Car() { // inutile non ?
        this.color = "gray";
        this.brand = "Kia";
        this.power = 7;
    }
    ...
}
```

Garbage collector : libérez la mémoire !

- Tout objet est créé par un appel explicite de l'opérateur « new » et toute manipulation s'effectue par référence
- Un objet conserve son emplacement mémoire tant qu'il est référencé, ce qui permet de mémoriser son état
- Dès qu'un objet n'est plus référencé, son emplacement mémoire va être récupéré
- La récupération de la mémoire est assurée par le Garbage Collector
- Le Garbage Collector est un processus dont l'exécution est indépendante de l'application qui libère la mémoire quand celle-ci n'est plus utilisée
- Il est possible de prévoir un traitement pour la libération (méthode « finalize() ») qui sera appelé implicitement pour le GC ou invoqué explicitement par le développeur.

Exemple

```java
public class Dummy {

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Bye, i'll be garbaged !");
    }
}

public class Demo {

    public static void main(String[] args) {

        Dummy dummy = new Dummy();
        dummy = null;
        System.gc();
    }
}
```

Affectation d'objet

- L'opérateur d'affectation est applicable sur les objets issus d'une même classe

```
class Demo {

    public static void main(String[] args) {

        Car my106 = new Car("green", "Peugeot", 5);
        Car myKia = my106;
        myKia.setColor("gray");
        my106.print();
        myKia.print();
    }
}
```

- Attention ! l'affectation est une affectation de référence !!
  - Pour faire une copie d'objet, il faut créer un constructeur par recopie ou redéfinir la méthode « clone »

Comparaison d’objet

- L’opérateur de comparaison est applicable sur les objets issus d’une même classe

```
class Demo {

    public static void main(String[] args) {

        Car my106 = new Car("green", "Peugeot", 5);
        Car myKia = new Car("gray", "Kia", 5);
        if (my106 == myKia) {
            System.out.println("Equality");
        } else {
            System.out.println("Different !");
        }
    }
}
```

- Attention ! La comparaison est une comparaison de référence !!
  - Si l’on veut faire une comparaison en fonction des attributs, il faut redéfinir la méthode « equals() »

La variable this

- La variable « this » est un argument implicite pour chaque méthode d'instance qui représente l'instance même
- Chaque méthode non statique contient implicitement un argument du type de la classe dans laquelle elle a été définie

```
public class Point {

    private int x;
    private int y;

    public int getX(Point this) {
        return this.x;
    }

}
```

Exemple

```java
public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point point) {
        this(point.x, point.y);
    }

    public boolean equals(Point point) {
        return x == point.x && y == point.y;
    }
}
```

```java
public class Demo {

    public static void main(String[] args) {

        Point p1 = new Point(17, 8);
        Point p2 = new Point(p1);
        if (p1.equals(p2)) {
            System.out.println("Equality");
        } else {
            System.out.println("Different");
        }
    }
}
```

Variable, méthode, instance classe

- Chaque instance de classe possède sa propre représentation, caractérisée par :
  - Les variables d'instance : attributs d'un objet, pour mémoriser l'état d'un objet
  - Les méthodes d'instance : reconnues par chaque instance de la classe
- On peut également définir :
  - Des variables de classe : globales à une classe, pour mémoriser l'état d'une classe
  - Des méthodes de classes : reconnues par la classe elle-même

Qualificatif « static »

- Le qualificatif « static » permet de définir des variables et méthodes de classe
- Les attributs statiques n'ont qu'une seule représentation en mémoire, partagée par l'ensemble des instances de la classe
- Les méthodes de classes peuvent être accédées directement par l'identificateur de classe (pas besoin d'instance de la classe)
- Une méthode statique ne peut pas utiliser un attribut non statique.

Exemple

```java
public class Counter {

    private static int nbInstances;

    public Counter() {
        this.nbInstances ++;
    }

    public static int howManyInstances() {
        return nbInstances;
    }

    public int getNbInstances() {
        return nbInstances;
    }

    public void finalize() {
        this.nbInstances --;
    }
}
```

Exemple

```java
public class Demo {

    public static void main(String[] args) {

        System.out.println(Counter.howManyInstances() + " instances");

        Counter counterOne = new Counter();
        System.out.println(Counter.howManyInstances() + " instances");
        System.out.println(counterOne.getNbInstances() + " from counterOne");

        Counter counterTwo = new Counter();
        System.out.println(Counter.howManyInstances() + " instances");
        System.out.println(counterOne.getNbInstances() + " from counterOne");
        System.out.println(counterTwo.getNbInstances() + " from counterTwo");
    }
}
```

Les classes de
base

Les classes de base Java

- Java définit un package « java.lang » qui est importé implicitement et qui forme le prolongement naturel du langage

- Ce package contient un ensemble de classes de base :
  - System // interraction avec le système
  - String // implémente le type chaine de caractères
  - Les wrappers // encapsulation des types primitifs
  - Math // traitements mathématiques
  - Class // Classe d’un objet

La classe String

- L’affectation d’une chaîne dans une String entraine sa création. C’est le mode de création préconisée
- Une String ne peut pas être modifiée : c’est une classe immuable.

```
public static void main(String[] args) {

    String one, two, three;
    one = new String("hello");
    two = "there";
    System.out.println(one + " " + two);
    three = one.toUpperCase();
    System.out.println(one + " " + three);
}
```

- Pour modifier une chaîne, il faut utiliser la classe « StringBuffer ».

Opérations sur les chaines

- Les opérateurs = et == s'appliquent sur les références uniquement
- L'égalité entre deux chaînes s'effectue donc avec la méthode « equals() »
- L'opérateur + permet la concaténation de deux chaînes

```java
public static void main(String[] args) {

    String one, two, three;
    one = new String("hello");
    two = new String("hello");
    three = one + two;
    System.out.println(three);
    if (one == two) {
        System.out.println("Ok ==");
    }
    if (one.equals(two)) {
        System.out.println("Ok equals() !");
    }
}
```

StringBuffer

- La classe StringBuffer permet de définir des chaînes modifiables

```
public static void main(String[] args) {

    StringBuffer buffer = new StringBuffer(20);
    buffer.append("Hello !\n");
    buffer.append("What's up?");
    System.out.println(buffer);
    buffer.setCharAt(7, ' ');
    System.out.println(buffer);
}
```

Les Wrappers

- Les wrappers sont des classes qui encapsulent les types primitifs
- La package « java.lang » fournit l'ensemble des wrappers suivant :
  - Integer
  - Long
  - Short
  - Float
  - Double
  - Byte
  - Boolean
  - Char

Service de base des Wrappers

- Constructeur permettant une instanciation à partir du type primitif
  - `Integer monInt = new Integer(10);`
- Constructeur permettant une instanciation à partir d’une String
  - `Integer monInt = new Integer("10");`
- Une méthode « [type]Value » pour fournir la valeur primitive représentée par l’objet
  - `int intValue = monInt.intValue();`

Service de base des Wrappers

- Un ensemble de méthodes de conversion
  - `Integer monInt = Integer.valueOf("10");`
  - `int intValue = Integer.parseInt("10");`
- Les classes Integer, Float et Double fournissent toutes les constantes
  - MAX_VALUE
  - MIN_VALUE
- Une méthode « equals() » pour l'égalité et une méthode « compareTo() » pour la comparaison.

Exercice : étude de cas

Étude de cas : Cahier des charges

- Nous souhaitons développer une application de gestion de comptes bancaires permettant à un gestionnaire de compte (ie employé de banque) de réaliser les opérations suivantes :
  - Créer un compte
  - Créditer un compte
  - Débiter un compte
  - Supprimer un compte
- Les fonctionnalités de bases :
  - La création d’un compte se fait à partir des informations suivantes :
    - Dépôt initial
  - À la création, une autorisation de découvert est définie de la façon suivante :
    - Découvert autorisé = dépôt initial

Étude de cas : cahier des charges (suite)

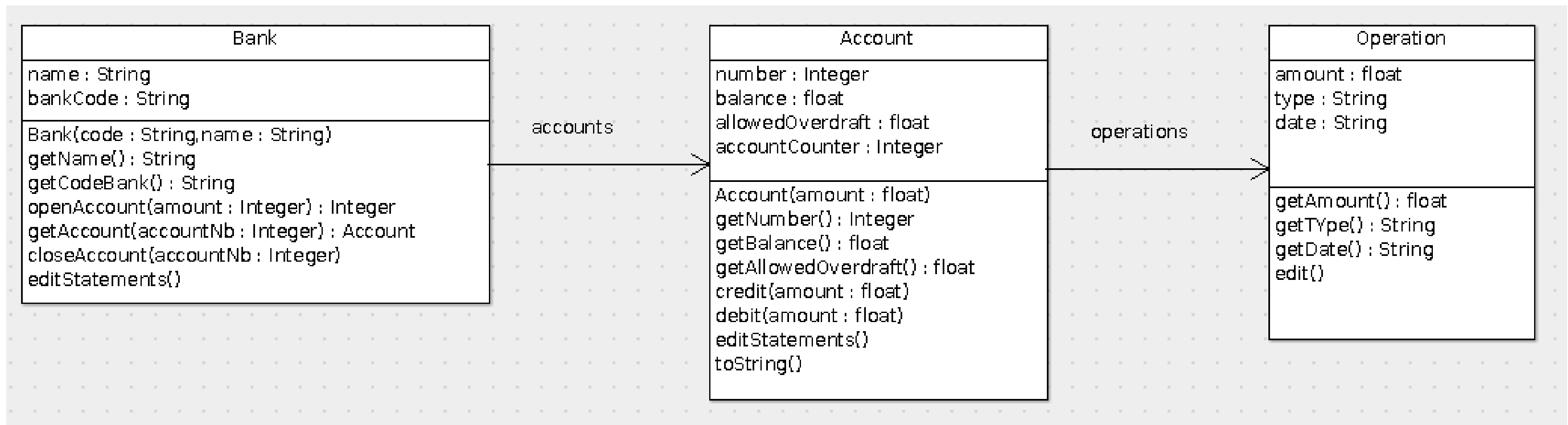
- L'opération « créditer un compte » permet d'ajouter un certain montant au solde d'un compte identifiable par son numéro. Cette opération n'est soumise à aucune contrainte métier (doit toujours être possible)
- L'opération « débiter un compte » permet de retirer un certain montant du solde d'un compte identifiable par son numéro. Cette opération n'est autorisé que si la contrainte suivante est respectée :
  - (solde + découvert autorisé) ≥ retrait
- L'opération « supprimer un compte » permet de supprimer au sein de la banque l'existence d'un compte identifiable par son numéro. Cette opération n'est réalisée que si le compte possède un compte positif ou nul.

Étude de cas : cahier des charges (suite)

- Chaque mouvement sur le compte (crédit ou débit) donne lieu à la création d'une opération associée au compte et tenant les informations suivantes :
  - Type de l'opération : crédit ou débit
  - Montant de l'opération
  - Date de l'opération
- Un gestionnaire de compte doit pouvoir lancer l'édition des relevés de tous les comptes. Un relevé de compte doit contenir les informations suivantes :
  - Nom de la banque et son code banque
  - L'état de chaque compte : N° de compte, solde, montant du découvert autorisé
  - Les opérations associées à chaque compte

Exercice n°2 :

- Implanter les classes Bank, Account et Operation à partir du modèle de classe ci-dessous, l'implantation de Bank est fournie à titre d'exemple.



- Dans un premier temps, nous considèrerons les liens entre les objets en relation unaire.
- Tester le bon fonctionnement des trois classes.

Héritage

Héritage & classe dérivée

- L'héritage permet de créer une classe par « extension » ou « spécification » d'une classe existante
- Chaque « classe fille » apporte un complément/spécification des caractéristiques supérieures
- Chaque caractéristique (attribut et méthode) définie sur une classe mère est accessible aux classes filles
- En java, si une classe D hérite d'une classe B, on parle de classe dérivée pour D et de classe de base pour B.

Exemple

```java
public class Person {

    private String name;
    private String fullname;
    private int age;

    public Person(String name, String fullname, int age) {
        this.name = name;
        this.fullname = fullname;
        this.age = age;
    }

    public void display() {
        System.out.println("I am " + fullname + " " + name + ", " + age + " years old");
    }

    public void birthday() {
        this.age ++;
    }
}
```

Exemple

```java
public class Student extends Person {
    private String classes;

    public Student(String name, String fullname, int age, String classes) {
        super(name, fullname, age);
        this.classes = classes;
    }

    @Override
    public void display() {
        super.display();
        System.out.print("Classes : " + classes);
    }
}

public class Demo {
    public static void main(String[] args) {
        Person person = new Person("Clarke", "Scott", 37);
        Student student = new Student("Henderson", "Dustin", 12, "Sciences");
        person.display();
        student.display();
        person.birthday();
        student.birthday();
        person.display();
        student.display();
    }
}
```

La classe dérivée « Student »

- Les caractéristiques héritées sont tributaires des conditions d’accès depuis la classe de base
- Les membres « private » de la classe « Person » ne sont pas accessibles par la classe « Student »

```
class Student extends Person {

    @Override
    public void display() {
        System.out.println("I am " + fullname + " " + name + ", "
                                + age + " years old"); // error
    }
```

- Les membres « public » de la classe « Person » sont eux accessibles par la classe « Student »

L’attribut d’accès « protected »

- Il est possible de rendre disponibles des attributs pour les classes dérivées via l’accès « protected »

```
public class Person {

    protected String name;
}

public class Student extends Person {
    private String classes;

    public void display() {
        System.out.println("I am " + name + ", in " + classes); // ok
    }
}
```

Le qualitatif « final »

- Sur un attribut, indique que le contenu ne doit pas évoluer (définition de constantes)

- Sur une méthode, indique que la méthode ne pourra être redéfinie dans les classes dérivées.

- Sur une classe, indique qu'aucune classe ne pourra dériver de celle-ci

Compatibilité d’instances

```java
public class Demo {
    public static void main(String[] args) {
        Person person1, person2;
        Student student1, student2;

        // Tout va bien ici
        person1 = new Person("Clarke", "Scott", 37);
        person2 = person1;
        person2.display();

        student1 = new Student("Henderson", "Dustin", 12, "Sciences");
        student2 = student1;
        student2.display();

        // Et si on mèle tout ça ?
        System.out.println("## Person1 from student1");
        person1 = student1;             // Ok : upstream compatibility
        person1.display();

        System.out.println("## Student1 from person1");
        student1 = person1;             // Nok : wrong type
        student1 = (Student) person1;   // Ok ?
        student1.display();

        System.out.println("## student2 from person2");
        student2 = (Student) person2;   // Ok ? why ?
        student2.display();
    }
}
```

Compatibilité d’instances

- La méthode appelée est toujours fonction de l’objet receveur (polymorphisme)
- L’opérateur « instanceof » permet de savoir si un objet appartient à une classe

```
boolean isIt = person1 instanceof Student;
System.out.println("is person1 instance of Student ?" + isIt);
```

- Il est possible de connaître le nom de la classe d’une instance

```
System.out.println("person1 is a " + person1.getClass().getCanonicalName());
System.out.println("person2 is a " + person2.getClass().getCanonicalName());
```

La classe Object

- Toute classe hérite implicitement de la classe « Object » : l'ensemble des classes des classes se présente sur la forme d'une hiérarchie en Java.
- La classe « Object » possède un ensemble de méthodes qui seront alors applicables sur chaque classe :
  - public boolean equals(Object o);
  - protected Object clone();
  - Public String toString();
  - public final Class getClass();
  - protected void finalize();

La classe Object

- La méthode equals() implémente une comparaison par défaut, basée sur les références (identique à l'opérateur '==')
  - `obj1.equals(obj2) ;  // true si obj1 et obj2 désignent le même objet`
- La méthode clone() fait une copie de l'objet. Chaque classe souhaitant bénéficier de ce service doit redéfinir cette méthode sous peine de produire l'Exception CloneNotSupportedException
  - `obj2 = obj1.clone() ;`
- La méthode toString() permet une conversion en chaine de caractères. Elle renvoie le nom de la classe suivi de l'opérateur « @ » lui même suivi de l'hachage de l'objet
  - `System.out.println( obj1.toString() ) ;`
- La méthode getClass() renvoie un objet de la classe Class qui représente la classe de l'objet
  - `String className = obj.getClass().getName() ;`

Exemple

```java
public class Point {

    protected int x;
    protected int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point p) {
        this.x = p.x;
        this.y = p.y;
    }

    public String toString() {
        return "(" + x + ";" + y + ")";
    }

    public boolean equals(Point p) {
        return this.x == p.x && this.y == p.y;
    }

    protected Point clone() {
        return new Point(this.x, this.y);
    }
}
```

```java
class Demo {
    public static void main(String[] args) {

        Point p1 = new Point(17, 8);
        Point p2 = new Point(p1);
        if (p1.equals(p2)) {
            System.out.println("p1 are p2 are equal : "
                                    + p1);
        } else {
            System.out.println("Not equals : " + p1
                                    + " VS " + p2);
        }
        Point px = p2.clone();
        System.out.println("here's px! " + px);

    }
}
```

Les classes
abstraites

Classe abstraite

- Une des idées maitresses de la programmation par objet consiste à construire de nouvelles classes sur la base de classes déjà existantes
- Cette technique de programmation ne révèle toute sa puissance que si les classes de base sont génériques.
  - Ainsi indépendantes de toute spécialisation, elles seront réutilisables dans un spectre plus large de situation
- Développer une classe générique consiste à implanter des méthodes en s’appuyant sur des comportements abstraits ; méthodes abstraites,
  - Méthodes abstraites définies par les classes dérivées
- La définition d’une classe abstraite se fait à l’aide du qualitatif « abstract »

Exemple

```java
public abstract class Vehicle {

    protected String color;
    protected int power;
    protected int nbPerson;
    protected int nbKm;

    public abstract void start();
    public abstract void move(int km);
    public abstract void stop();

    public void transport(int nbPerson, int km) {
        this.nbPerson = nbPerson;
        this.start();
        this.move(km);
        this.stop();
    }
}
```

Classe abstraite : restrictions

- Une classe abstraite n'est utilisable que comme classe de base par d'autres classes
- Toute tentative d'instanciation d'une classe abstraite provoque une erreur

```
Vehicle myCar = new Vehicle() ; // Error
```

- Toute classe ayant au moins une méthode abstraite est considérée comme abstraite et doit être définie comme telle avec le qualitatif « abstract »
- Une classe dérivée d'une classe abstraite doit fournir une définition pour chaque méthode déclarée abstraite, ou redéclarer ces méthodes abstraites

Exemple

```java
public class Bus extends Vehicle {

    @Override
    public void start() {
        System.out.println("Bus start");
    }

    @Override
    public void move(int km) {
        this.drive(km);
    }

    @Override
    public void stop() {
        System.out.println("Bus stop");
    }

    public void drive(int km) {
        System.out.println("Drive for " + km + " !");
        this.nbKm += km;
    }
}
```

```java
public class Demo {
    public static void main(String[] args) {
        Bus aBus = new Bus();
        aBus.transport(50, 20);
    }
}
```

Étude de cas : évolution du cahier des charges

- Le système doit maintenant être capable de gérer deux types de compte: des comptes standards (comptes courants) et des comptes d'épargne :
  - Un compte d'épargne possède les mêmes caractéristiques qu'un compte standard mais permet en plus de produire des intérêts en fonction d'une valeur de taux définie à sa création.
  - Le gestionnaire de compte doit pouvoir à, tout moment, générer le calcul des intérêts applicables à tous les comptes épargnes de la banque.
  - Pour chaque compte d'épargne, le résultat de ce calcul est ajouté au solde ainsi qu'à la propriété « Cumul des intérêts » permettant de conserver le montant des bénéfices financiers produits par le compte.
- Les attributs spécifiques à un compte épargne sont donc les suivants :
  - Valeur du taux
  - Cumul des intérêts

Exercice n°3

- Implanter la classe SavingsAccount et l'intégrer à la solution précédente en modifiant le moins possible les classes existantes (Bank et Account).
- Désormais, on aura dans Bank une opération createSavingsAccount
  - Note : à ce stade, Bank possède toujours un seul account, même lors de la création d'un SavingAccount
- Tester le bon fonctionnement d'ensemble.

Parenthèse : Les collections (java.util)

- Une collection est une entité qui gère un ensemble d’objets d’une classe donnée
- Framework pour les collections depuis jdk1.2
- Trois types fondamendaux de collection :
  - List : collection ordonnée, duplication d’objet possible
  - Set : collection sans double, les objets sont uniques
  - Map : collection de paires clé/valeur
- Tous les types de collection supportent les opérations de bases : ajouter, retirer et itérer sur les objets de la collection

Les Tableaux dynamiques : ArrayList

- « ArrayList » est un tableau dont la taille peut évoluer en fonction des besoins
- Dérive de « List »
- Comme toute classe, il vient avec un ensemble de constructeurs
  - ArrayList() ;
  - ArrayList(int capacity) ;
  - ArrayList(Collection c) ;
- L’ajout d’élément dans un ArrayList se fait par une des méthodes « add » :
  - boolean add(Object obj) ;
- La récupération se fait à l’aide de la méthode « get »
  - Object get(int index) ;
- Il est possible de spécifier le type d’éléments qui que contiendra l’ArrayList lors de sa création :
  - `ArrayList<Person> group = new ArrayList<>() ;`

ArrayList : exemples

```java
public class Demo {
    public static void main(String[] args) {

        Point p1 = new Point(17, 8);

        ArrayList list = new ArrayList();
        list.add("one");
        list.add(p1);
        list.add(10);
        list.add(3.14159d);
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }

        ArrayList<String> messages = new ArrayList<>();
        messages.add("Hello");
        messages.add("How are you");
        messages.remove(1);
        messages.add("How are you ?");
        messages.add("Great !");
        for (String message : messages) {
            System.out.println(message);
        }
    }
}
```

Exercices n°4

- Faire évoluer le modèle de données en appliquant maintenant les relations 0..* à l'aide de la classe ArrayList :
  - Bank contient une liste de Account : accounts
  - Account contient une liste de Operations : operations
- Les méthodes «openAccount », «openSavingsAccount »,« debit » et « credit » ajoutent désormais un élément à la liste concernée.

Les interfaces

Interfaces

- Une interface est un contrat
- Lorsqu’un objet client d’une interface collabore avec celle-ci, il ne connaît pas son type réel mais il sait comment le « manipuler »
- But des interfaces :
  - Diminuer le couplage entre les classes
    - L’interface permet de ne présenter au client que « ce qui l’intéresse »
  - Limiter les impacts sur des « clients » lors de modifications des fournisseurs
  - Se protéger des variations
    - Un objet client d’un interface peut collaborer avec n’importe quel type d’objet pourvu qu’il réalise l’interface

Interface : Exemple de mise en exemple

- Soit une interface Musicable spécifiant les comportements que doit respecter tout musicien

```java
public interface Musicable {
    public void playMusic();
}
```

- Un guitariste étant un musicien, il devra donc respecter l'interface

```java
public class Guitarist implements Musicable {

    @Override
    public void playMusic() {
        System.out.println("You play the guitar on the MTV !");
    }
}

public class Drummer implements Musicable {

    @Override
    public void playMusic() {
        System.out.println("I shoulda learned to play them drums");
    }
}
```

Interface : Exemple de mise en exemple

- Une classe client de l’interface « Musicable » :

```
public class Orchestra {
    protected List<Musicable> musicians = new ArrayList<>();

    public void addMusician(Musicable musician) {
        this.musicians.add(musician);
    }

    public void play() {
        for (Musicable musician : musicians) {
            musician.playMusic();
        }
    }
}
```

Héritage multiple

- Java ne propose pas l'héritage multiple, MAIS une classe peut hériter d'une classe et implanter une ou plusieurs interfaces :

```
public class Drummer extends Person implements Musicable {

    @Override
    public void playMusic() {
        System.out.println(this.fullname + " shoulda learned to play them drums");
    }
}
```

- Une classe implémentant plusieurs interfaces :

```
public class Clown extends Person implements Musicable, Acrobat ... { }
```

Étude de cas : nouvelle évolution du besoin

- Chaque compte doit maintenant être associé à un titulaire :
  - Il existe deux types de titulaires : des personnes et des sociétés
  - De futures versions devront pouvoir introduire de nouveaux types de titulaires (des associations par exemple) en impactant le moins possible l'existant
- Une personne est caractérisée par :
  - Nom, prénom, adresse
  - Le nom et le prénom définissent le nom complet du titulaire lorsque ce titulaire est incarnée par une personne, l'adresse d'une personne définissant l'adresse du titulaire.
- Une société est caractérisée par :
  - Sa raison sociale, son n° de siret, son siège social
  - La raison sociale et le n° de siret d'une société définissent le nom complet du titulaire lorsque ce titulaire est incarnée par une société, le siège social d'une société définissant l'adresse du titulaire.

Exercice n°5

- Identifier les impacts qu'engendre cette nouvelle notion sur les classes existantes

- Implanter l'interface Holder et les classes Person et Company, les intégrer à la solution précédente
  - Un relevé de compte exploite l'interface Holder pour définir le destinataire du relevé (nom et adresse)

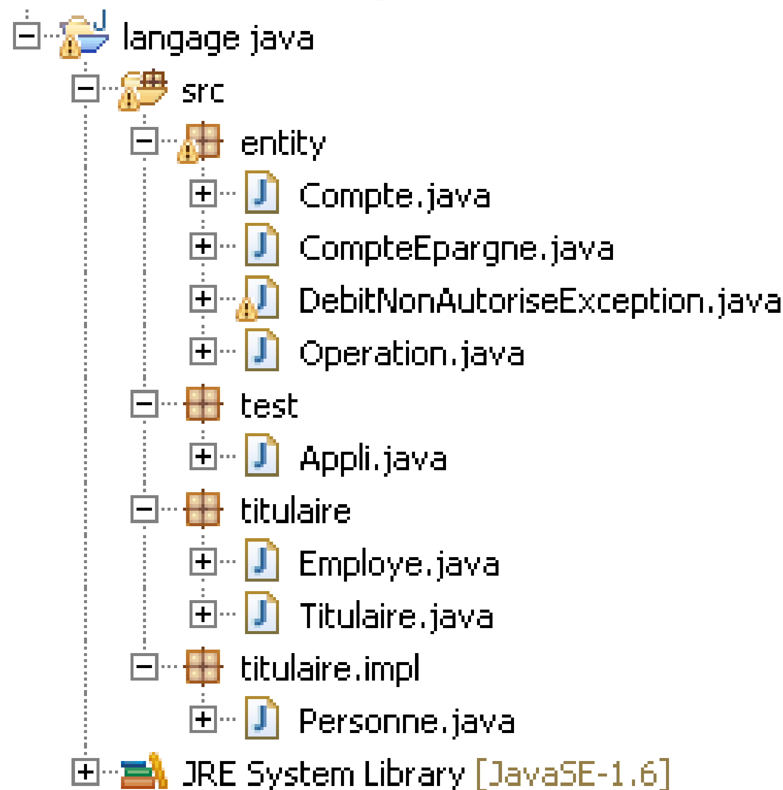
Les packages

Packages

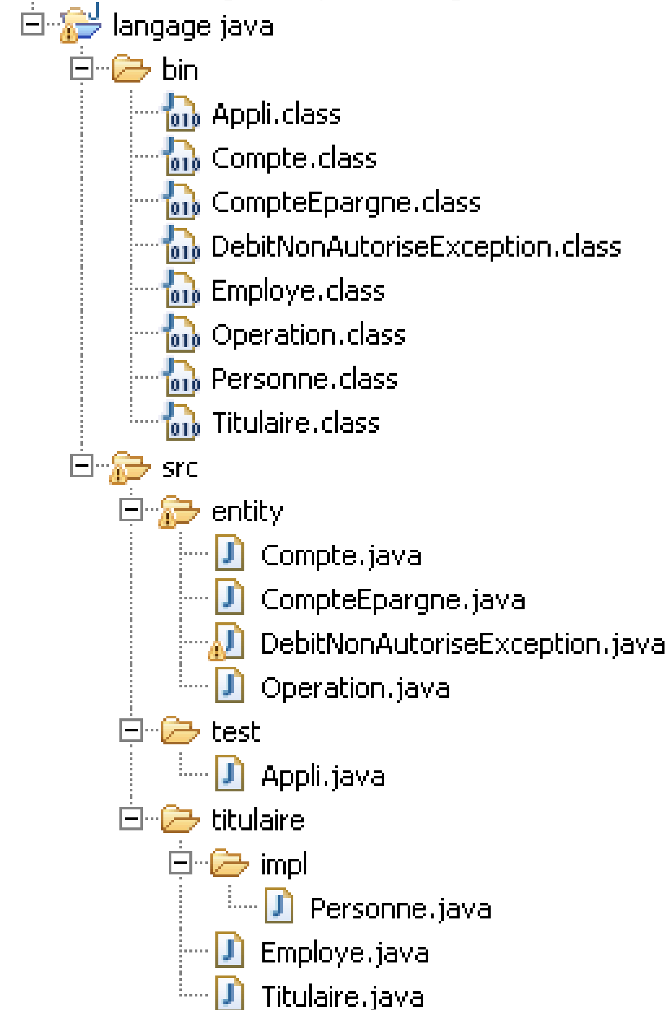
- Les packages permettent de regrouper les classes par « affinité »
  - Construisent l'organisation logique d'une application Java
  - Définissent les espaces de noms dans lesquels sont définies les classes
  - Constituent des entités d'encapsulation, de gestion de version, des taches de développement, …
- Du point de vue des sources Java, un package représente un répertoire
  - Le mot clef réservé « package » permet de définir le package d'une classe
  - Le mot clef « import » permet de faire référence à une classe en dehors de son propre package

Exemple d'oranisation en packages

- Vue logique



- Vue physique



Les packages : mise en oeuvre

```
formation-java [formation-java-dev3]
  .idea
  src
    main
      java
        com.codelutin
          examples
          exercice
            Driver
            Journey
          Demo
```

```java
package com.codelutin.exercice;

import java.util.ArrayList;
import java.util.List;

/**
 * @author ymartel (martel@codelutin.com)
 */
public class Journey {

    private int distance;

    private List<Driver> drivers;
```

Organisation des API standards

- Les API standards Java sont elles-mêmes organisées en packages
- Quelques packages essentiels de la librairie standard :
  - java.applet : les applets sur le web
  - java.awt : interfaces graphiques, images et dessins
  - java.io : entrées/sorties
  - java.lang : chaînes de caractères, interraction avec l'OS, threads, …
  - java.util : structure de données classiques
  - Java.swing : interfaces graphiques « light weigth »
  - Java.sql : fournit le package JDBC

Exemple : le package java.util

- Propose des classes représentant les collections
- Pour manipuler les collections, il faut importer le package java.util
  - import java.util.* ; //Utilisation de l'ensemble du package
  - import java.util.ArrayList ; // Uniquement ArrayList
- util est un sous répertoire de java qui contient le code compilé (byte-code) de l'ensemble des classes qu'il implémente
  - ArraysList.class, HashMap.class, …
- Le fichier ArrayList.java se présente sous la forme suivante :

```
package java.util;

public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable { … }
```

Règles de nommage

- La création et l'utilisation de packages sont basées sur des règles de nommage strictes
- Le système de nommage est basé sur l'organisation hiérarchique de ses packages
- L'identificateur d'un package doit correspondre au répertoire ou à un chemin d'accès dans lequel se trouve la ou les classe(s)
- L'utilisation d'un package se fait en spécifiant le répertoire ou le chemin d'accès contenant la ou les classe(s) que l'on veut importer
- La partie interface d'un fichier source (class public) doit posséder le même identificateur que le fichier source

Exemple

- Création d’un package pour la classe « Person »

```
package code.lutin.pers;
public class Person {
```

- Le fichier Person.java compilé va générer un fichier Person.class

```
import code.lutin.pers.Person;
// import code.lutin.pers.*;

class Demo {
    public static void main(String[] args) {
        Person person1;
        person1 = new Person("Clarke", "Scott", 37);
        person1.display();
        person1.birthday();
        person1.display();
    }
}
```

Exercice n°6

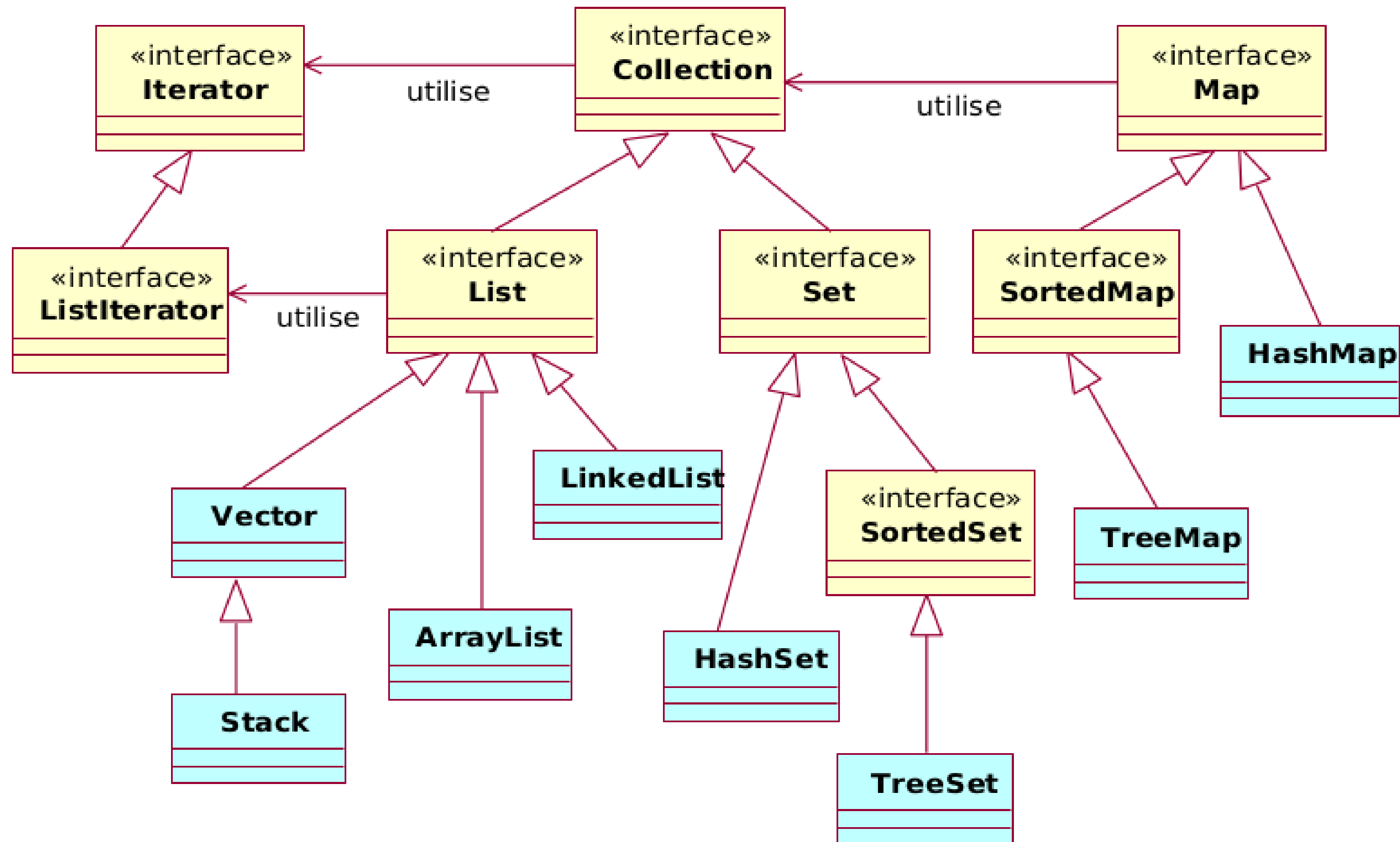
- Restructuration du projet banque en packages dans le but de :
  - Limiter les dépendances
  - Favoriser le développement collaboratif

Les collections

Les collections Java

- Une collection (parfois appelée conteneur) est un objet qui regroupe de multiples références d'objets
- Un framework de collections est une architecture unifiée de représentation et de manipulation des collections
- La framework de collections Java consiste en :
  - Des interfaces
  - Des collections concrètes (structure de données réutilisables)
  - Des algorithmes (fonctionnalités réutilisables)

Présentation du framework : diagramme UML



L'interface Collection

- Utilisée pour manipuler n'importe quelle collection concrète à partir d'une interface de services générale

```
public interface Collection<E> {
    boolean add(E o);
    boolean addAll(Collection<E> c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean equals(Object c);
    boolean isEmpty();
    Iterator<E> iterator();
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    int size();
    Object[] toArray();
}
```

L’interface List

- Collection ordonnée (parfois appelée Séquence) qui peut contenir des éléments dupliqués (redondance)
- Elle vient compléter l’interface collection en ajoutant des méthodes spécifiques pour les listes

```
public interface List<E> extends Collections {
    boolean add(int index, E o);
    E remove(int index);
    E get(int index);
    E set(int index, E o);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    List<E> subList(int fromIndex, int toIndex);
}
```

La classe ArrayList

- Implémentation la plus courante de l'interface List

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    private static final int DEFAULT_CAPACITY = 10;
    transient Object[] elementData; // non-private to simplify nested class access
    private int size;

    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }

    public ArrayList() {
        this.elementData = EMPTY_ELEMENTDATA;
    }

    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }
}
```

Exemple

```
import java.util.ArrayList;
import java.util.List;

public class Demo {
    public static void main(String[] args) {

        List<Point> points = new ArrayList<>();
        points.add(new Point(10, 20));
        points.add(new Point(20, 40));
        points.add(new Point(30, 50));
        for (Point point : points) {
            point.toString();
        }
    }
}
```

Recherche d'un élément

- La méthode contains() permet de rechercher si un élément est présent dans la collection :

```
public class Demo {
    public static void main(String[] args) {

        List<Point> points = new ArrayList<>();
        Point p1 = new Point(10, 20);
        points.add(p1);
        points.add(new Point(20, 40));
        points.add(new Point(30, 50));
        if (points.contains(p1)) {
            System.out.println("ok pour p1");
        } else {
            System.out.println("non ok pour p1!");
        }
        if (points.contains(new Point(20, 40))) {
            System.out.println("ok pour (20;40)");
        } else {
            System.out.println("non ok pour (20;40)!");
        }
    }
}
```

Redéfinition de la méthode equals

- La classe Point redéfinit la méthode equals :

```
public class Point {

    protected int x;
    protected int y;

    // ...

    public boolean equals(Object obj) {
        if (!(obj instanceof Point)) {
            return false;
        }
        Point point = (Point) obj;
        return ((this.x == point.x) && (this.y == point.y));
    }

}
```

Les itérateurs

- Un itérateur permet de parcourir une collection sans se soucier de la nature réelle du conteneur. Ainsi on développe des algorithmes de parcours constant alors que la nature du conteneur varie
- Le parcours générique d'une collection s'effecture de la manière suivante :
  - Appel de la méthode **iterator()** sur le conteneur pour renvoyer un iterator
  - Récupérer l'objet suivant la séquence grâce à sa méthode **next()**
  - Vérifier s'il reste encore d'autres objets dans la séquence via la méthode **hasNext()**
- On a également la possibilité d'enlever le dernier élément renvoyé par l'itérateur avec la méthodes **remove()**

Exemple

```java
public class Demo {
    public static void main(String[] args) {

        List<Point> points = new ArrayList<>();
        points.add(new Point(10, 20));
        points.add(new Point(20, 40));
        points.add(new Point(30, 50));

        Iterator<Point> iterator = points.iterator();
        while (iterator.hasNext()) {
            Point point = iterator.next();
            System.out.println(point.toString());
            if (point.x == 20) {
                iterator.remove();
            }
        }
        System.out.println("new collection has " + points.size() + " elements :");
        for (Point point : points) {
            System.out.println(point.toString());
        }
    }
}
```

L’interface Set

- Correspond à un groupe d’objets qui n’accepte pas deux objets égaux au sens de « equals » (comme les ensembles en mathématiques)
- l’interface Set n’ajoute aucune méthode à la clase Collection. Elle ne spécifie que les restrictions attendues sur les méthodes de base (unicité des objets)
- La méthode **add** n’ajoute pas l’élément si un élément égal (au sens du equals) est déjà dans l’ensemble (la méthode renvoie alors false)

```
public class Demo {
    public static void main(String[] args) {

        Set s = new HashSet();
        String tab[] = {"a", "b", "c", "b"};
        for (int i = 0; i < tab.length; i++) {
            Boolean insert = s.add(tab[i]) ;
            if (! insert) {
                System.out.println("Duplication détectée : " + tab[i]);
            }
        }
        System.out.println(s.size() + " mots dans le set " + s);
    }
}
```

La classe TreeSet

- C’est un ensemble trié. Il garantit que les éléments sont rangés dans leur ordre naturel
- Pour effectuer le tri, le TreeSet a besoin d’utiliser une méthode de comparaison
- Une relation d’ordre sur des objets peut être définie en implémentant l’interface *Comparable*
- l’interface Comparable consiste en une seule opération ;

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Les classes implémentant cette interface peuvent être triées automatiquement

L’interface Comparator

- Une relation d’ordre naturelle n’est pas toujours suffisante :
  - Ordonner des objets à travers une autre relation d’ordre
  - Ordonner des objets qui n’implémentent pas l’interface Comparable
- Un comparateur (interface *Comparator*) peut être utilisé dans le cas où la relation d’ordre naturelle ne suffit pas :

```
public interface Comparator {
    public int compare(Object o1, Object o2);
}
```

- Les collections effectuant un tri sur leurs éléments peuvent être instanciées en fournissant un objet comparateur spécifique :

```
new TreeSet(Comparator c);
```

Exemple d’implantation de l’interface Comparable

```java
public class Revue implements Comparable {

    protected String title;
    protected int number;

    public Revue(String title, int number) {
        this.title = title;
        this.number = number;
    }

    public String getTitle() {
        return title;
    }

    public int getNumber() {
        return number;
    }

    @Override
    public String toString() {
        return title + " " + number;
    }

    @Override
    public int compareTo(Object o) {
        return title.compareTo(((Revue) o).getTitle());
    }
}
```

Exemple de mise en œuvre de l'interface Comparable via un TreeSet

```java
import java.util.Collection;
import java.util.Iterator;
import java.util.TreeSet;

public class Demo {

    public static void main(String[] args) {
        Collection<Revue> revues = new TreeSet<>();
        // les éléments seront triés sur l'ordre naturel
        // basé sur le surcharge de l'interface Comparable

        revues.add(new Revue("Geo", 1));
        revues.add(new Revue("CQ", 2));
        revues.add(new Revue("France Football", 3));
        revues.add(new Revue("Linux Pratique", 4));

        Iterator<Revue> iterator = revues.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next()); //toString
        }
    }
}
```

L'interface Map

- l'interface Map correspond à un groupe de couples clés-valeurs

```
public interface Map {
    void clear();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Set entrySet();
    Set keySet();
    Object get(Object key);
    boolean isEmpty();
    Object put(Object key, Object value);
    Object putAll(Map map);
    void remove(Object key);
    Collection values();
    int size();
}
```

- Une clef repère un et un seul objet. Deux clefs ne peuvent être égales au sens de equals

Les HashMap

- Les classes qui implantent l'interface Map sont :
  - **HashMap** : table de hachage garantissant un accès en temps constant
  - TreeMap : arbre ordonné suivant l'ordre naturel des éléments (interface Comparable) ou fonction d'un objet Comparator fourni au constructeur
- Exemple

```
public class Demo {

    public static void main(String[] args) {
        Map<String, Person> map = new HashMap<>();
        map.put("Clarke", new Person("Clarke", "Scott", 37));
        map.put("Henderson", new Student("Henderson", "Dustin", 12, "Sciences"));
        map.get("Clarke").display();
    }
}
```

Utilitaires pour les collections : la classe **Collections**

- La classe **Collections** fournit un ensemble d'algorithmes, sous forme de méthodes statiques, permettant de travailler sur toute instance des classes impémentant l'interface Collection (tri, recherche, copie…) :

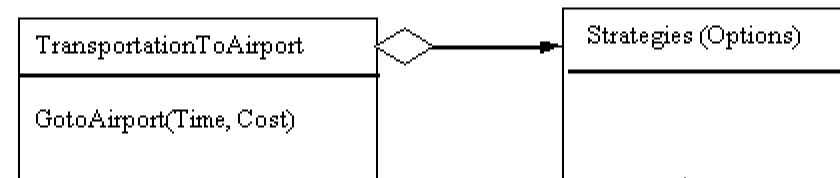
```java
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Demo {

    public static void main(String[] args) {

        String stringTab[] = {"chaine 3", "chaine 1", "chaine 2"};

        List<String> stringList = Arrays.asList(stringTab);
        Collections.sort(stringList);
        for (String stringElement : stringList) {
            System.out.println(stringElement);
        }
    }
}
```

Exercice n°7

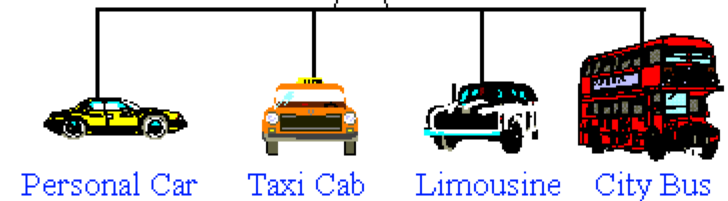
- Faire évoluer le modèle de données pour que les collections garantissent l’unicité de leurs éléments.

Introduction aux Design Patterns : Strategy

- Strategy est un pattern utilisé lorsque :
  - De nombreuses classes associées ne diffèrent que par leur comportemant
  - Pour configurer une classe avec un comportement parmi plusieurs (politique)
  - On a besoin de plusieurs variantes d'algorithme
  - Un algorithme utilise des données que les clients ne doivent pas connaître



- Une alternative au sous classement…



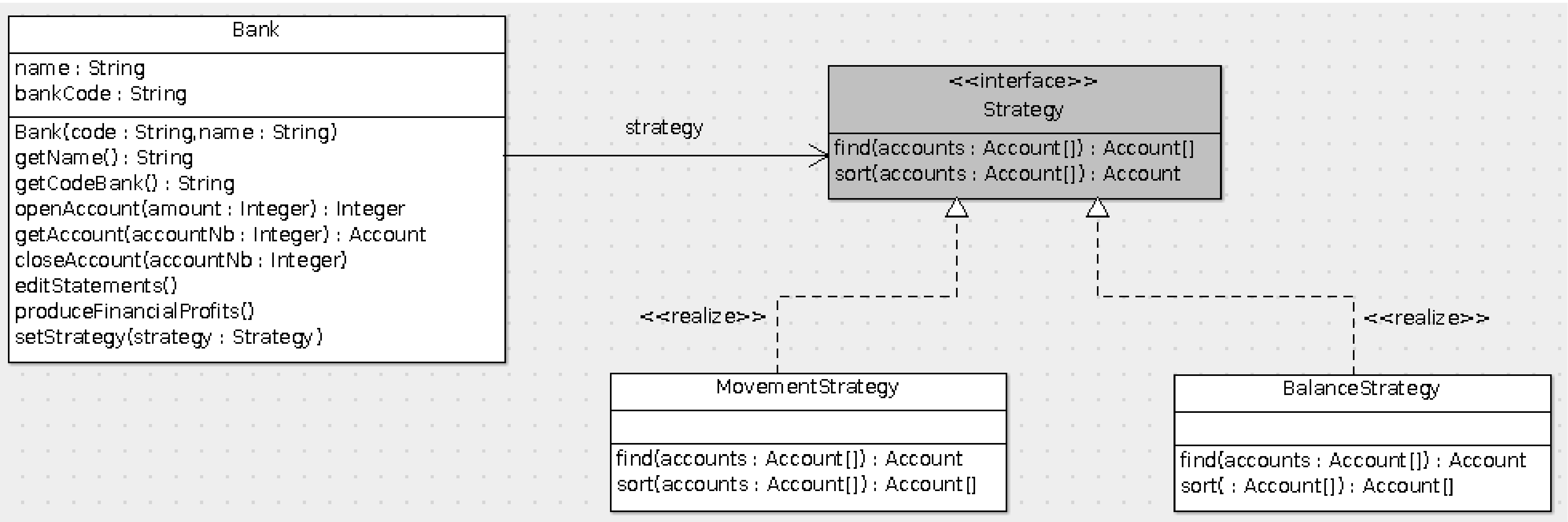
Strategy : les participants

- Les classes et/ou objets participant dans ce pattern sont :
  - **Strategy** (**SortStrategy**)
    - Déclare une interface commune pour tous les algorithmes supportés
  - **ConcretStrategy** (**QuickSort**, **ShellSort**, **MergeSort**)
    - Implante les algorithmes conformément à l'interface de la Strategy
  - **Context** (**SortedList**)
    - Est configuré avec un objet concreteStrategy
    - Maintient une référence sur un objet Strategy
    - Peut définir une interface qui permet à la stratégie d'accéder à ses données
    - Fournit des services « haut niveau » implantés en exploitant la stratégie courante

Exercice n°8

- On veut pouvoir rechercher un compte selon différents critères :
  - Celui qui a le solde le plus élevé
  - Celui qui a effectué le plus de mouvements
  - …
- Aussi effectuer les relevés de tous les comptes selon ces mêmes critères
  - Du solde le moins élevé au solde le plus élevé
  - Des dormants aux plus actifs
  - …
- Construire des stratégies de gestion de comptes en utilisant l'interface Comparator<T>

Exercice n°8 : modèle UML

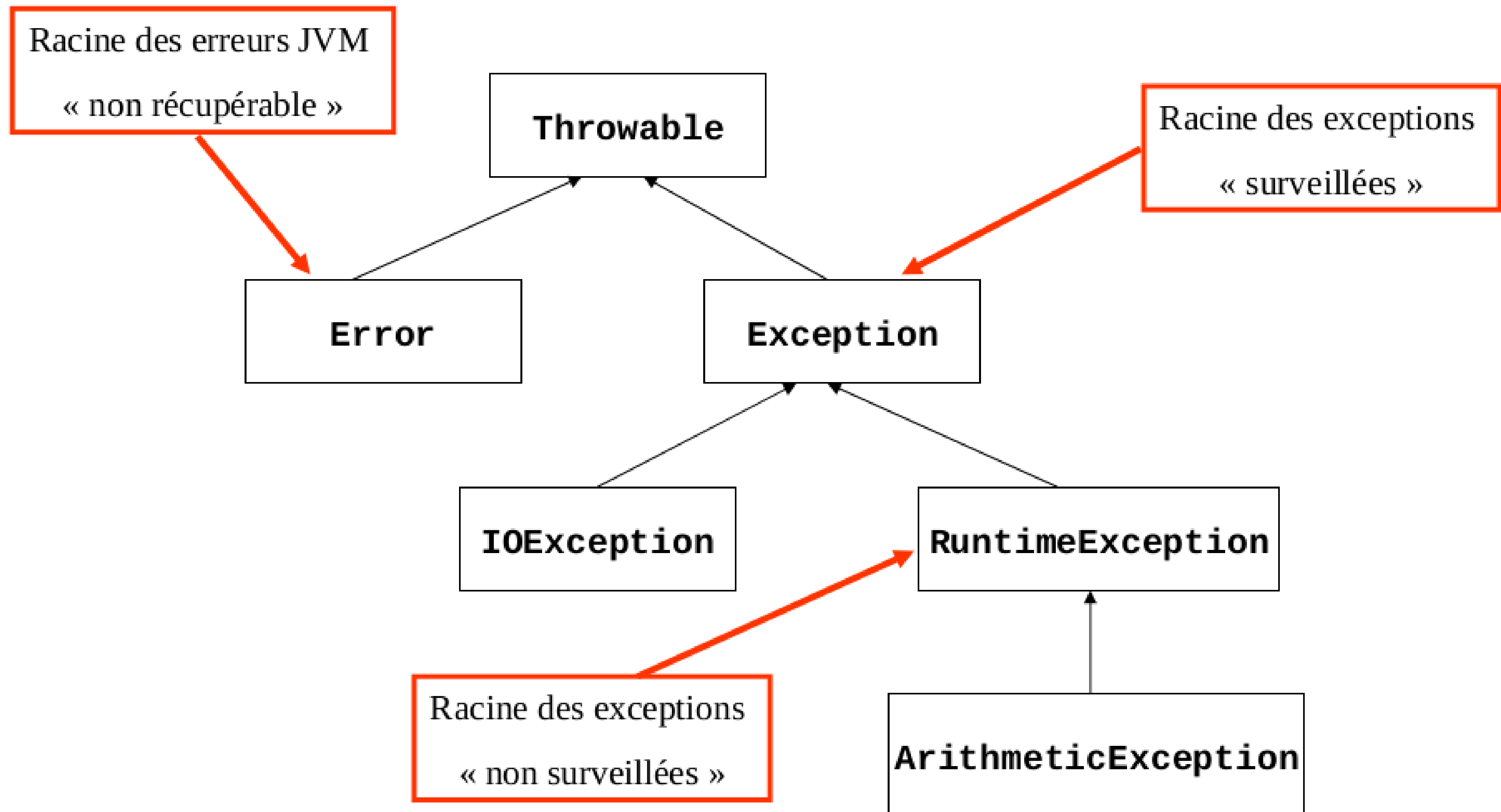


Les exceptions

Les exceptions

- Le mécanisme d’exception définit un standard pour la gestion des erreurs. Il permet de rendre la gestion des erreurs plus lisible et plus régulière.
- Lorsqu’une situation d’erreur est détectée, il faut :
  - Abandonner l’action en cours,
  - Signaler cette situation d’erreur,
  - Exécuter les actions appropriées
- Les erreurs peuvent être classées en deux catégories : les erreurs surveillées et les erreurs non surveillées

Hiérarchie standard des exceptions Java



Quelques exceptions et erreurs standards en Java

- Exceptions « non surveillées » standards les plus rencontrées :
  - java.lang.NullPointerException
  - java.lang.ClassCastException
  - java.lang.ArrayIndexOutOfBoundsException
- Exceptions « surveillées » standards les plus rencontrées :
  - java.sql.SQLException
  - java.io.IOException
  - Java.lang.ClassNotFoundException
- Les erreurs standards les plus courantes :
  - Java.lang.OutOfMemory
  - Java.lang.StackOverflow

Exemple de mise en œuvre d’une exception « non surveillée »

```
public class Appli {

    public static void main(String[] args) {

        String[] mots = new String[4];

        mots[0] = "Ceci est";
        mots[1] = "un";
        mots[2] = "test";
        for (String mot : mots) {
            System.out.println("Mot de " + mot.length() + " lettres");
        }
    }
}
```

- Résultat d’exécution :

```
Mot de 8 lettres
Mot de 2 lettres
Mot de 4 lettres
Exception in thread "main" java.lang.NullPointerException
	at fr.inra.agrosyst.services.performance.indicators.Appli.main(Appli.java:25)

Process finished with exit code 1
```

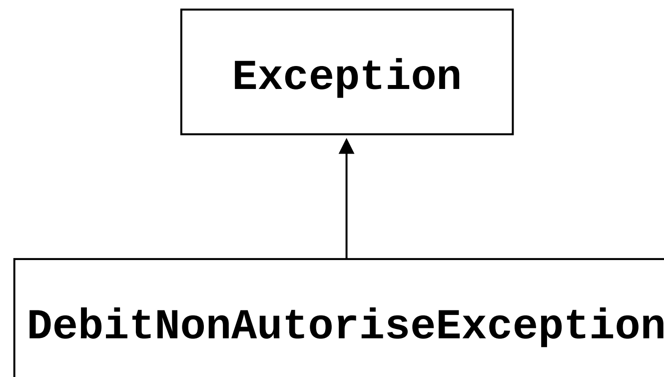
Exception « surveillée »

- L’appel de méthodes pouvant lever une ou des exceptions « surveillées » oblige l’appelant de les prendre en compte :
  - Traiter l’exception,
  - La repropager au contexte appelant supérieur
- Structure de code de prise en compte d’une exception

```java
public void foo() {
    try {
        // traitement pouvant générer une Exception
    } catch (Exception e) {
        // traitement de l'exception
    } finally {
        // faire quelque chose dans tous les cas ...
    }
    // suite du traitement non soumis à exception ...
}
```

Exception « métier »

- Une exception « métier » caractérise le non respect d'une contrainte de gestion sur un objet « métier »
  - Exemple : le débit d'un montant supérieur au solde courant d'un compte est interdit
- Une exception « métier » est toujours de type « surveillé »
  - Définit par une classe d'exception spécifique
  - Doit donner lieu à un traitement correctif adapté
- Exemple :



Lever une exception

- l'action « **throw** » (mot réservé) permet de lever une exception et la propager à l'appelant

```
public class Sailboat {
    public void maneuver() throws Exception {
        // ..
        if (problem) {
            throw new Exception("Boat in distress!");
        }
    }
}
```

Attraper une exception

- Pour attraper une exception, il faut englober le code susceptible de la lancer dans un bloc **try** et traiter l'exception dans un bloc **catch** :

```
public class Transat {
    public static void main(String[] args) {
        Sailboat fujicolor = new Sailboat();
        try {
            fujicolor.maneuver();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

- Si aucune erreur n'intervient pendant l'exécution du bloc **try**, le contrôle est passé à l'instruction qui suit le dernier bloc **catch**.

Créer des exceptions métier

- Java offre la possibilité de créer des exceptions métier par héritage de la classe Exception :

```java
public class SosException extends Exception {

    protected float longitude, latitude;

    public SosException(float longitude, float latitude, String message) {
        super(message);
        this.longitude = longitude;
        this.latitude = latitude;
    }
}

public class Sailboat {
    public void maneuver() throws SosException {
        if (problem)
            throw new SosException(50.34f, 50.12f, "Boat in distress");
    }
}
```

Spécification d’exceptions

- Lorsqu’on utilise une classe, il est fondamental de connaître les méthodes soumises à des exceptions

- Le langage permet de spécifier les exceptions pouvant être générées lors de l’exécution des méthodes
  - void method() throws ExceptionA { … new ExceptionA() ;… }
  - void method() throws ExceptionA, ExceptionB { … }

- La spécification des exceptions énumère tous les types dexceptions pouvant être leves durant l’exécution

Traitement des exceptions

- Un bloc **try** est immédiatement suivi d’au moins un bloc **catch**
- Le paramètre du **catch** indique le type d’exception traité
- On peut définir autant de blocs **catch** que de types d’exceptions possibles
- Les blocs **catch** sont parcourus dans leur ordre d’apparition jusqu’à concordance entre les types de leur argument et le type de l’objet exception généré
- Un **catch** contrôle les exceptions de sa classe ainsi que celles de ses classes dérivées
- Dans un bloc **catch**, Java offre la possibilité de relancer la dernière exception générée à l’aide du mot réservé **throw**
- Depuis java 7, un bloc **catch** peut concerner plusieurs exceptions en les séparant grâce à l’opérateur « | »

Finaliser le traitement

- Le langage met à disposition une instruction « finally » qui sera exécutée en sortie de bloc **try**, en toutes circonstances :
  - Exécution du bloc try sans levée d’exception
  - Exécution du bloc try avec levée d’une exception

```
try {
    // traitement pouvant générer une Exception
} catch (Exception e) {
    // traitement d'exception
} finally {
    // traitement effectué en toutes circonstances, facultatif
}
```

Exercice n°9

- Mise en œuvre d’une exception métier NotEnoughMoneyException lorsqu’un opération de débit est supérieure au solde + découvert autorisé :
  - Lever l’exception dans la méthode « debit » de la classe Account

Les entrées-sorties

Les entrées-sorties

- Le package java.io fournit un ensemble de classes permettant de gérer les entrées-sorties :
  - Les streams de caractères et streams binaires,
  - Les streams à accès direct ou séquentiel,
  - Les streams avec tampon de données.

Les entrées-sortie en mode binaire (octet)

- Les classes d’entrées-sorties en mode binaire héritent de deux classes abstraites : InputStream pour les entées et OutputStream pour les sorties
- Les principales classes sont les suivantes :
  - FileInputStream, FileOutputStream : lecture ou écriture séquentiele de données dans un fichier
  - BufferedInputStream, BufferedOutputStream : lecture ou écriture des données à l’aide d’un tampon
  - PipedInputStream, PipedOutputStream : permet d’établir une connexion entre un stream d’entrée et un stream de sortie pour la communication inter thread
  - ObjectInputStream, ObjectOutputStream : lecture ou écriture de données représentant des objets (sérialisation)
  - PrintStream : écriture de données avec conversion en octets en fonction du système hote (pour écrire au format texte les types primitifs Java).

Les entrées-sorties en mode caractère

- Les classes d'entrées-sorties en mode caractères héritent de deux classes abstraites : Reader pour les entrées et Writer pour les sorties
- Les principales clases sont les suivantes :
  - FileReader, FileWriter : lecture ou écriture séquentiel de caractères dans un fichier
  - BufferedReader, BufferedWriter : lecture ou écriture de caractères à l'aide d'un tampon
  - PipedReader, PipedWriter : permet d'établir une connexion entre un stream d'entrée et un stream de sortie
  - InputStreamReader, OutStreamReader : permet la conversion d'un stream de données binaire en stream de caractères

Exemple

```
public class File {
    public static void main(String[] args) {
        FileReader input = null;
        FileWriter output = null;
        try {
            input = new FileReader("sourceFile");
            output = new FileWriter("destFile");
            char c;
            while ( (c= (char) input.read()) != -1 ) {
                output.write(c);
            }
        } catch (IOException ioe) {
            System.out.printf(ioe.getMessage());
            ioe.printStackTrace();
        } finally {
            if (input != null) {
                try {
                    input.close();
                } catch (IOException e) {
                    System.out.println("error during input close");
                    e.printStackTrace();
                }
            }
            if (output != null) {
                try {
                    output.close();
                } catch (IOException e) {
                    System.out.println("error during output close");
                }
            }
        }
    }
}
```

Exemple

```
public class File {
    public static void main(String[] args) {
        String str;
        try {
            FileReader input = new FileReader("sourceFile");
            BufferedReader in = new BufferedReader(input);
            while ((str = in.readLine()) != null) {
                System.out.println(str);
                ;
            }
            in.close(); // pas bien !
        } catch (IOException ioe) {
            System.out.printf(ioe.getMessage());
            ioe.printStackTrace();
        }
    }
}
```

Accès direct

- La classe RandomAccessFile permet d'accéder directement à des données

```
public class Demo {

    public static void main(String[] args) {

        try {
            RandomAccessFile fic;
            fic = new RandomAccessFile("file1.txt", "rw");
            fic.seek(10);
            while (fic.getFilePointer() < fic.length()) {
                System.out.println(fic.readLine());
            }
        } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
            ioe.printStackTrace();
        }
    }
}
```

La sérialisation

- Java met à disposition des streams permettant d'enregistrer des objets sur disque. Cette opération (sérialisation) permet de rendre les obkets persistants.

- Un objet voulant être sérialisé doit implanter l'interface **Serializable**. Cette interface ne possède aucune méthode, elle est utilisée en tant que marqueur.

- Lorsque l'objet est sauvegardé, tous les objets auxquels il fait référence le sont aussi (ils doivent donc implanter eux aussi l'interface **Serializable**).

- Si l'objet hérite d'une autre classe qui elle n'est pas sérialisable, il faut que cette classe aie un constructeur qui ne prend aucun paramètre.

Exemple

```java
import java.io.Serializable;

public class Car implements Serializable {
    protected int km;
    protected int fuel;
    protected Motor motor;

    public Car(String brand) {
        this.motor = new Motor(brand);
    }

    public int getKm() { return km; }

    public void setKm(int km) { this.km = km; }

    public int getFuel() { return fuel; }

    public void setFuel(int fuel) { this.fuel = fuel; }

    public Motor getMotor() { return motor; }

    public void setMotor(Motor motor) { this.motor = motor; }
}
```

Exemple

```java
import java.io.Serializable;

public class Motor implements Serializable {
    protected int power;
    protected String brand;
    protected static int nbInstance; // Car counter

    public Motor(String brand) {
        this.brand = brand;
    }

    public int getPower() { return power; }

    public String getBrand() { return brand; }

}
```

- Une exception NotSerializableException est générée lorsque l'on veut sérialiser un objet issu d'une classe qui n'implémente pas **Serializable**.

Sauvegarde d’objet

- La sérialisation d’un objet est effectuée lors de l’appel de la méthode « writeObject() » sur un objet implantant l’interface **ObjectOutput** (ObjectOutputStream).
  - Void writeObject(Object odj) ;
- Par défaut, tous les champs de l’objet sont sauvés.
- Les attributs statiques ne sont pas sérialisés.
- Chaque fois qu’un objet est sauvé dans un flux, un objet *handle* unique pour ce flux est également sauvé. Ce *handle* est attaché à l’objet dans une table de hashage.
- Lors d’une même sérialisation globale, chaque fois que l’on demande de sauver un objet déjà présent dans le flux, seul le *handle* est sauvé. Ceci permet de casser les circularités.

Restitution d'objet

- La désérialisation d'un objet est effectuée lors de l'appel de la méthode « readObject() » sur un objet implantant l'interface ObjectInput (ObjectInputStream)

```
Object readObject() ;
```

- L'objet récupéré est une copie de l'objet sauvé.

Exemple

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class TestSerialisation {
    public static void main(String[] args) throws Exception {
        Car car = new Car("Audi");
        car.setFuel(50);
        car.setKm(3000);
        FileOutputStream outputFile = new FileOutputStream("garage.ser");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
        outputStream.writeObject(car);
        outputStream.close();
        car = null;
        System.gc();
        FileInputStream inputFile = new FileInputStream("garage.ser");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile);
        car = (Car) inputStream.readObject();
        inputStream.close();
        System.out.printf(car.getMotor().getBrand() + " " + car.getFuel());
    }
}
```

La sérialisation : compléments

- Il est possible d'empêcher la sauvegarde de certaines données à l'aide du mot réservé **transient** :

  transient int carburant ;

- La redéfinition de la méthode writeObject dans sa classe permet de sérialiser les attributs statiques qui ne le sont pas implicitement :

```
private void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
    s.writeInt(this.nbInstance); // static attribute
}
```

- Il est possible de définir son propre traitement de restitution en redéfinissant la méthode readObject dans sa classe :

```
private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    this.nbInstance = s.readInt(); // static attribute
}
```

La gestion de version

- Un objet ne peut petre restitué que s’il a été sauvegardé avec la même version de l’application
- Java propose un mécanisme permettant de « Identifier » le problème de la lecture d’un objet d’une classe qui a évoluée depuis que l’objet a été sauvegardé.
- Il faut dans ce cas identifier la version de la classe :

```java
private static final long serialVersionUID = 1138186247452163451L;
```

- Une valeur est assignée automatiquement au SUID lorsque la classe est sérialisée.
- l’utilisatire serialver du jdk permet de récupérer le SUID

Exercice n°10

- Rendre tous les objets de l’application sérialisables.

JDK et environnement d'exécution

Le jdk

- Le JDK (Java Development Kit) distribué gratuitement par Oracle fournit l’ensemble des outils de base pour le développement d’applications Java :
  - Un compilateur : **javac**
  - Une jvm : **java**
  - Un testeur d’applet : **appletviewer**
  - Un déboggueur : **jdb**
  - Un créateur de paquet pour distribution : **jar**
  - Une système de génération de documentation : **javadoc**
  - Un signeur de jar : **jarsigner**
  - …

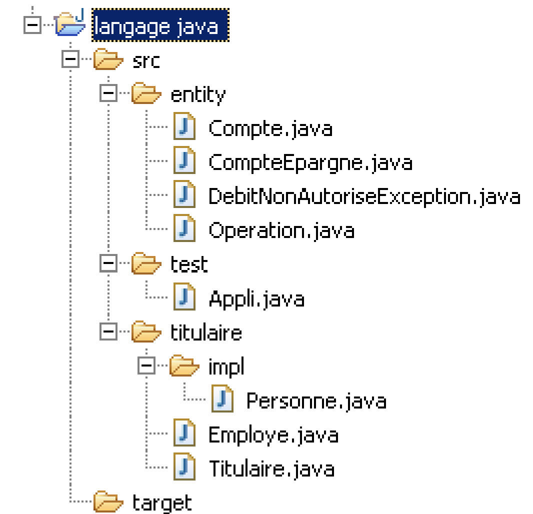
Outils du JDK : le compilateur

- Le compilateur javac :
  - Transforme le code source en byte code exécutable par la JVM sur n’importe quelle plateforme ...
  - … pour une version donnée du JDK

- Syntaxe : javac <option> <source files>
  - **-g** : génère les informations de debug
  - **-verbose** : trace les activités du compilateur
  - **-classpath** <path> : répertoires des fichiers .class
  - **-sourcepath** <path> : répertoires des fichiers Java
  - **-d** <directory> : répertoire de génération des .class
  - **-source** <release> : effecture une vérification de compatibilité des sources avec une version spécifique du JDK
  - **-target** <release> : génère les .class pour une version spécifique de JVM

Le compilateur javac : mise en œuvre

Organisation physique avant compilation



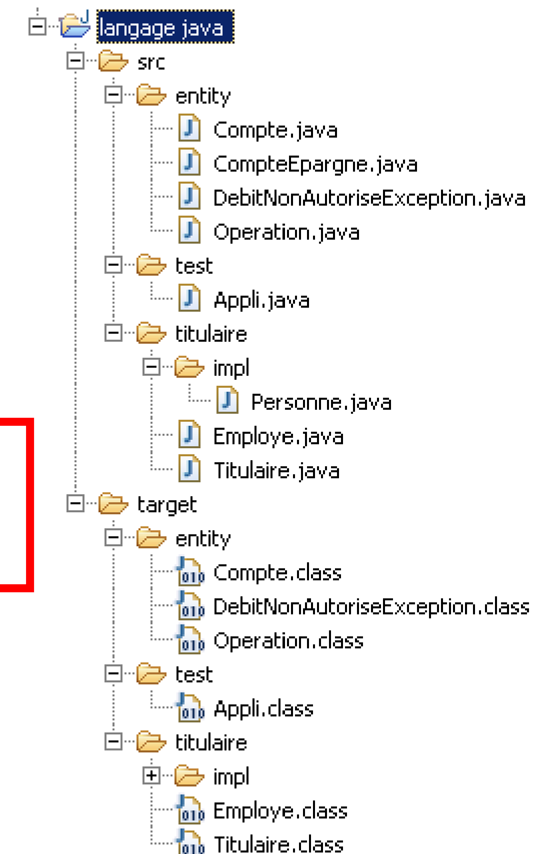
- Compilation

  cd ~/workspace/java

  javac -sourcepath src -d target

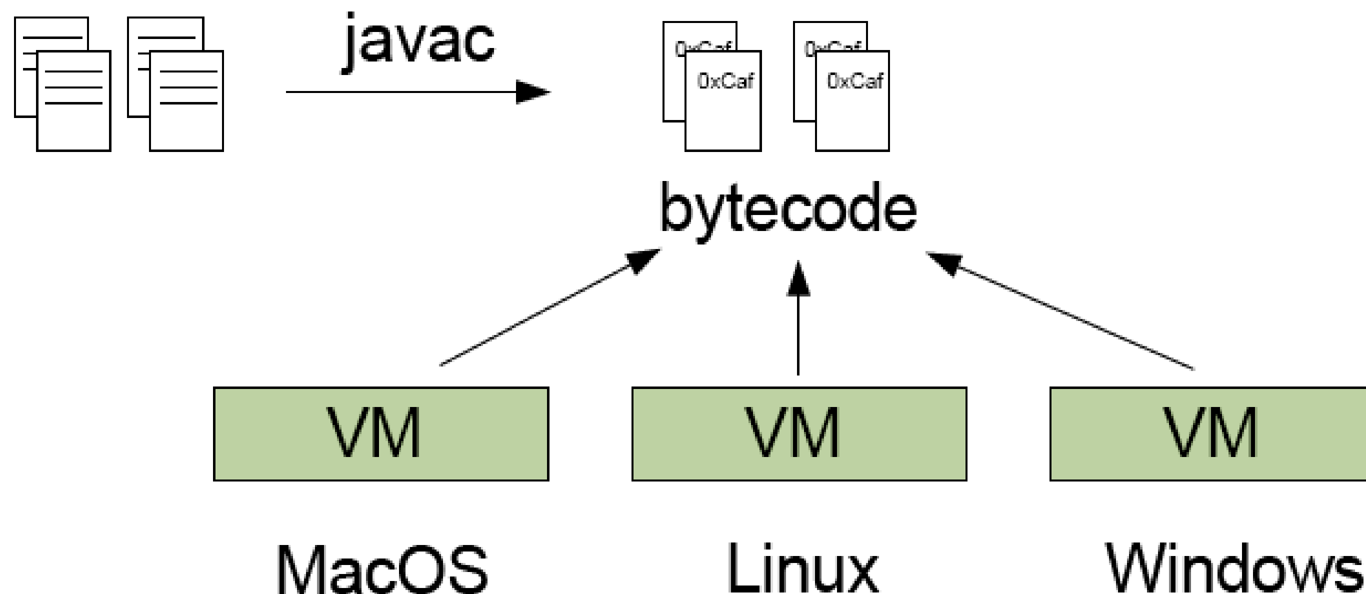
- La compilation est transitive !

Organisation physique après compilation



Outils du JDK : la JVM

- Le compilateur Java génère un « byte code », langage assembleur de la JMV Java. Ce code est ensuite interprété.
- La JVM assure la portabilité entre différents environnements d'exécution (OS)



JVM et la sécurité

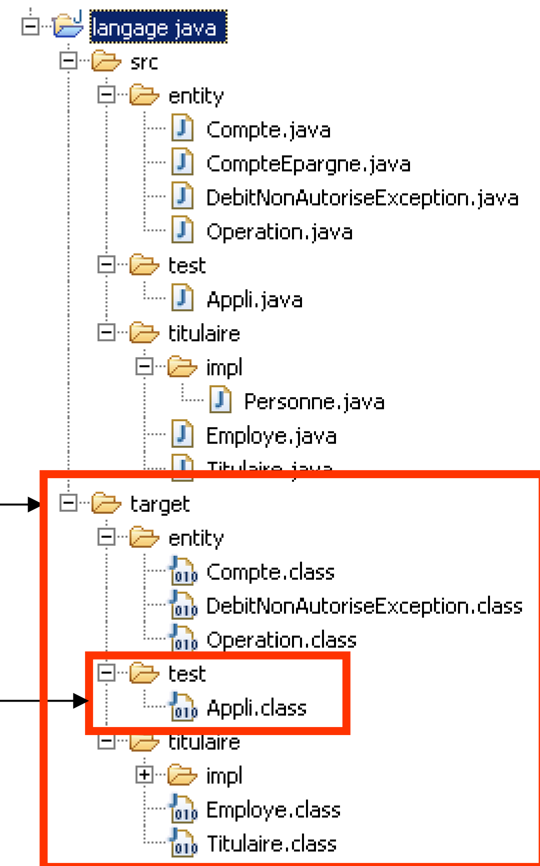
- Prise en charge dans l'interpréteur
- Trois couches de sécurité :
  - Vérifier : vérifie le byte code
  - Class Loader : responsable du chargement des classes
  - Security Manager : controleur d'accès aux ressources
- Les code peut par ailleurs être signé
  - Cas des applets nécessitant l'accès aux ressources locales du client (certificat)

JMV : mise en œuvre

- Syntaxe :
  - Pour exécuter un .class : **java [-options] class[args…]**
  - Pour exécuter un JAR : **java [-options] -jar jarfile [args…]**
- Principales options
  - **-cp <class search path of directories and zip/jar files>** : fournit la liste de répertoires et d'archives (JAR et ZIP) séparés par « ; » et dans lesquels rechercher les .class
  - **-D<name>=<value>** : valorise une variable d'environnement
  - **-version:<value>** : requière la version spécifiée
  - **-showversion** : affiche la version de la JVM
  - **-ea[:<packagename>..|:<classname>]** : active les assertions
  - **-da[:<packagename>..|:<classname>]** : désactive les assertions

JVM : mise en œuvre

- Exécution de la classe Appli
  - Java -classpath target test.Appli



- Résultat d'exécution
  - Débit impossible : solde=10.0 montant demandé=100.0

Les outils du JDK : la commande jar

- L'archiveur de classes :
  - Outil standard pour consruire/lire des archives de type JAR
  - Équivalent à la notion de librairie dans d'autres langages
- Syntaxe similaire à la commande « tar » unix :
  - jar {ctxui}[vfm0Me][fichier-jar][fichier-manifeste] [point-entrée] [-C rép] fichiers …
- Principales options :
  - **-c** : crée une nouvelle archive
  - **-x** : extrait les fichiers nommés (ou tous les fichiers) de l'archive
  - **-u** : met à jour l'achive existante
  - **-f** : spécifie le nom du fichier archive
  - **-m** : inclut les informations de manifeste à partir du fichier de manifeste spécifié
  - **-C** : passe au répertoire spécofoé et inclut le fichier suivant

Mise en œuvre de la commande JAR

- Mise en œuvre :
  - Jar czf appli.jar -C target\ .

- Exécution du jar
  - Java -classpath appli.jar test.Appli
  - Résultat : débit impossible : solde=10.0 montant demandé=100.0

Créer un jar exécutable

- Nécessite un fichier MANIFEST définissant la classe du jar contenant le point d'entrée (méthode « main »)
- Exemple de fichier MANIFEST

  Manifest-Version : 1.0

  Main-Class : test.Appli

- Production du JAR
  - Jar -cvfm appli.jar appli-manifest -C target\ .
- Exécution du JAR
  - Java.jar appli.jar

Autre outils du JDK

- l'interpréteur d'applets
  - Appletviewer permet de visualiser l'éxécution d'un applet
  - Syntaxe : jappletviewer monApplet.class
- Le générateur de documentation :
  - JavaDoc a pour but de créer une documentation uniforme au format HTML à partir du code
    - Grace aux commentaires prévus à cet effet (/**)
    - La documentation de l'API standard Java est de la JavaDoc

Les versions de Java

- 1.0 : version initiale lancée en 1995
- 1.1 : 1997, ajout de jdbc pour les connexions aux bases de données, fichier Jar, instrospection et sérialisation entre autres
- 1.2 : 1998, Playground : JDBC 2 et compilateur JIT
- 1.3 : 2000, Kestrel : Grandes améliorations de performances
- 1.4 : 2002, Merlin : JDBC 3, API de Logging et Java Web Start
- 5.0 : 2004, Tiger : Grande améliorations du langage (boucle foreach, génériques, …)
- 6.0 : 2006, Mustang : meilleure intégration avec le système d’exploitation, avec les classes Desktop et Systrays
- 7.0 : 2011, Dolphin : ajout du multicatch, chaines dans switch, opérateur « diamond » pour les génériques
- 8.0 : 2014, Spider : lambdas, streams, nouvelles API Date/Time
- 9.0 : 2017 : Jigsaw (modularité), jshell, fabrique pour collections
- 10.0 : 2018
- 11.0 : 2018, fin de Corba, JavaFX

Vos remarques?

02 • 40 • 50 • 29 • 28

www.codelutin.com