Concepts objets, exceptions et collections

Programmation avancée – Java

F5 – ISIMA 2020/2021

Olivier Goutet
o.goutet@openium.fr

6 octobre 2020

Plan

Plan

Créer une instance

- Demander la mémoire au système
  - Opérateur `new`
- Appeler le constructeur
  - `Classe instance = new Classe(parametres);`
- Manipulation de ~~pointeurs~~ références ?
- Valeur spéciale `null`
  - Si création impossible `instance == null`
  - Ou pas encore affectée

Détruire une instance

- Pas de destruction manuelle
- Destruction automatique par la JVM
  - Ramasse-miettes (Garbage Collector)
- Plus de fuites de mémoire ?
  - Static, Singleton
  - non surcharge de `equals()` & `hashCode()`
  - Anonymous inner class non statiques
  - File ou connexion JDBC non fermée
  - Boucle infinie
- Méthode `finalize()`
  - Ressemble au destructeur C++
  - Peut ne pas être appelée (si gc non exécuté)
  - Très très lents
  - Peuvent ne jamais être appelés
  - => Éviter les finalizers [1]

1. Effective Java : Item 7

Contenu de la classe

- Attributs
  - d'instance
  - de classe
- Méthodes
  - d'instance
  - de classe
- Constructeur(s)
- Instructions au chargement de la classe

Attributs

- Listés n'importe où (début de classe préférable)
- Valeurs par défaut (!= variables locales)
- D'instance

```
String chaine;
int entier;
```

- De classe
  - Accessibles sans créer d'objet
  - Initialisés à la déclaration ou bloc spécial

```
static int compteur = 0;
```

Méthodes

- D'instance

```
private String nom1(double d);
protected int nom2(int a, int b);
```

- De classe
  - Accessibles sans créer d'objet
  - Ne peut PAS accéder aux attributs d'instance

```
static type_retour nomMethode(parametres);
```

Modificateurs de méthode

```
public static final int methode(double d);
```

- `public` Méthode publique, visible/pouvant être appelée par tout le monde
- `static` Méthode de classe
- `final` Méthode non redéfinissable
- `int` Type de retour
- `double d` Paramètre(s)

Constructeur

- Initialiser les attributs d'un nouvel objet
- Syntaxe différente d'une méthode
  - Porte le même nom que la classe
- Constructeur par défaut
  - fourni automatiquement si pas d'autres constructeurs
- Surcharge de constructeur
  - Appel de constructeurs != avec paramètres
- Pas d'héritage de constructeur

Exemple complet

```
public class Cours {
  int nb_etudiants;

  public Cours() {
    nb_etudiants = 0;
  }
  public Cours(int n) {
    nb_etudiants = n;
  }
  public int getNbEtudiants() {
    return nb_etudiants;
  }
  public boolean isPassionnant() {
    return true;
  }
}
```

Encapsulation

- Sécurisé les données
- Données privées
- Accesseur est une méthode publique qui donne l'accès à une variable d'instance privée (get / set)

```
private int valeur = 13;

public int getValeur(){ return(valeur); }
public void setValeur(int val) {valeur = val;}
```

Au chargement de la classe

- Instructions spécifiques exécutées au chargement de la classe dans la JVM
  - Pas à l'instanciation d'objet

```
class Exemple {
  static int[] tab;
  static {
    // execute au chargement de la classe
    tab = new int[4];
    for(int i=0; i<4; ++i) tab[i] = i+1;
  }
}
```

Héritage

- B hérite de A
- Conditions
  - A doit être visible (publique ou même package par ex)
  - A n'est pas finale
- B hérite de tous les membres **protégés** et **publics** de A sauf les constructeurs
  - Les membres **privés** ne sont pas transmis
- B n'hérite que d'une **seule** classe directe
- Toute classe hérite de `java.lang.Object`

Héritage

```
public class B extends A {
  public B() {
    super(); // appel du constructeur de A
    // initialisations specifiques
  }
}
```

- **super** : ce qui vient de la classe mère

```
super.methode();
super.attribut;
super.super.attribut // illegal
```

- **this** : concerne l'objet courant

Référence this

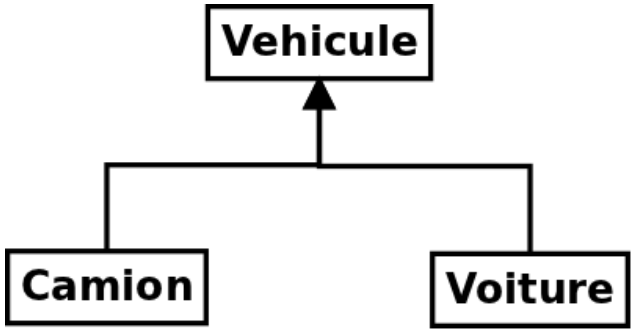
```
public class A {
  String chaine1, chaine2;

  public A() {
    chaine1 = "CHAINE1";
    chaine2 = "CHAINE2";
  }

  void methode1(String chaine1, String c) {
    this.chaine1 = chaine1;
    chaine2 = c;
  }

  void methode2() {
    methode1("e","f");
    this.methode1("","");
  }
}
```

Noms qualifiés

```
class A {
  protected int attribut;
}

public class B extends A {

  protected double attribut;

  public void toto() {
    attribut              // est un double
    this.attribut         // est un double
    super.attribut        // est un int
    ((A)this).attribut    // est un int
    ((B)this).attribut    // est un double
  }
}
```

Hériter

- Écrire une classe Vehicule qui reprend les caractéristiques communes des classes Voiture et Camion
- On va modifier les classes pour tester le polymorphisme...



Hériter

```
public class Vehicule {
  public void afficher() {
    System.out.println("Vehicule");
  }
  public static void main(String[] param) {
    Vehicule v = new Vehicule();
    Voiture  w = new Voiture();
    Camion   c = new Camion();
    Vehicule z = new Voiture();
    Voiture  i = new Vehicule();
  }
}

class Voiture extends Vehicule {
  public void afficher() {
    System.out.println("Voiture");
  }
}
class Camion extends Vehicule {
}
```

Appeler les méthodes `afficher()` des objets.

Hériter

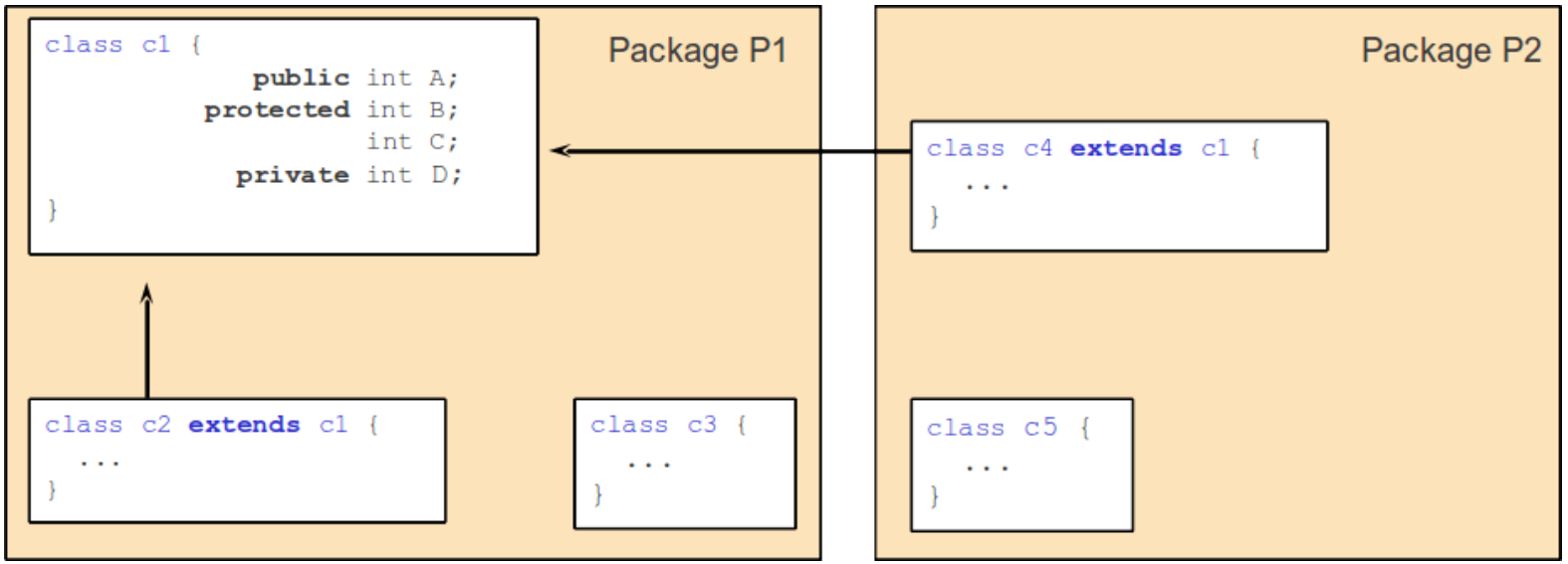
```
class Voiture extends Vehicule {
  public void afficher() {
    System.out.println("Voiture");
  }
  public void special() {
    System.out.println("special");
  }
}

public class Vehicule {
  public void afficher() {
    System.out.println("Vehicule");
  }
  public static void main(String[] param) {
    Vehicule v = new Voiture();
    v.afficher();
    v.special();
    ((Voiture)v).special();
    ((Camion)v).afficher(); // defini precedemment
  }
}
```

Que se passe-t'il ?

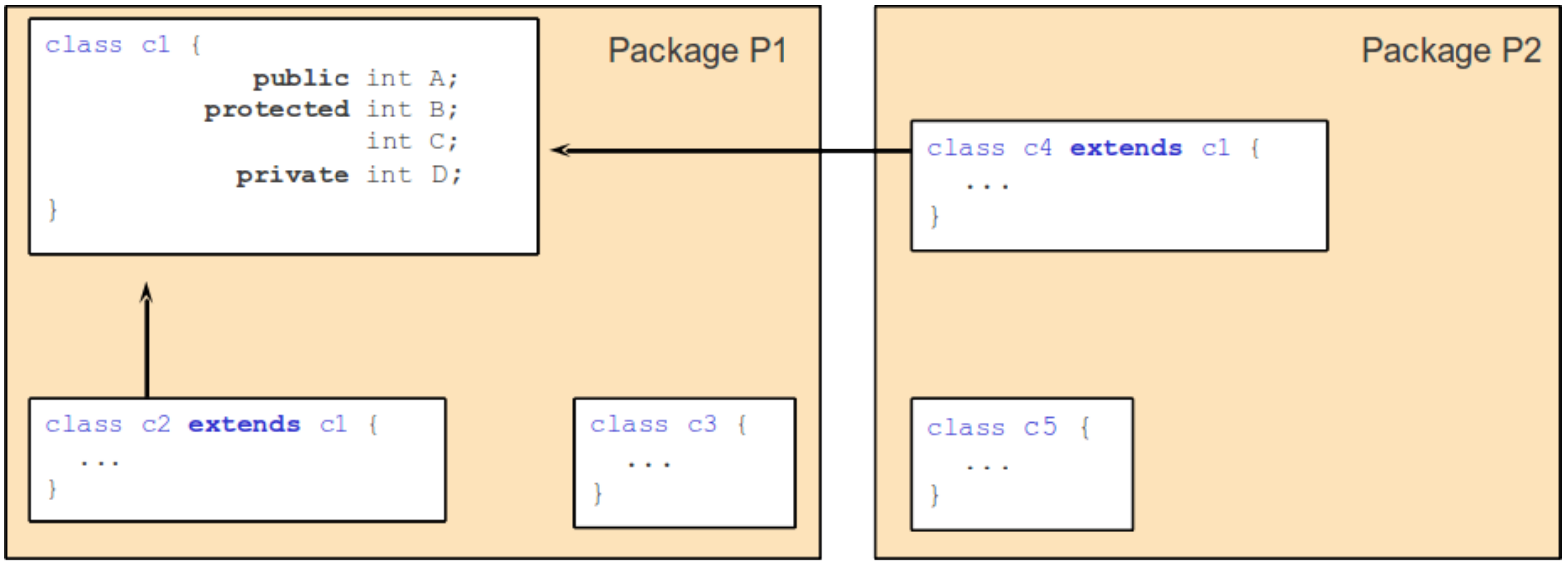
Niveaux d'accès

- public
  - tout le monde
- private
  - même classe
- protected
  - même package
  - sous-classe d'un package différent
- package - (par défaut)
  - sorte de friend du C++
  - DANGEREUX

Encapsulation des membres



Encapsulation des membres



	A	B	C	D
Accessible par c2	o	o	o	-
Accessible par c3	o	-	o	-
Accessible par c4	o	o	-	-
Accessible par c5	o	-	-	-

Méthodes virtuelles ou finales ?

- Méthodes virtuelles
  - Par défaut
  - Construction d'une table de méthodes pour une hiérarchie
  - Recherche dans cette table (lenteur ?)
- Méthodes finales
  - Non redéfinissables dans les classes filles
  - Plus rapides que les méthodes virtuelles
  - Conseil : accesseurs en final (attributs publics ?)

Méthodes et classes abstraites

- Méthode abstraite
  - sans implémentation
  - mot-clé `abstract` (modificateur)
- Classe abstraite
  - Non redéfinissables dans les classes filles
  - Plus rapides que les méthodes virtuelles
  - Conseil : accesseurs en final

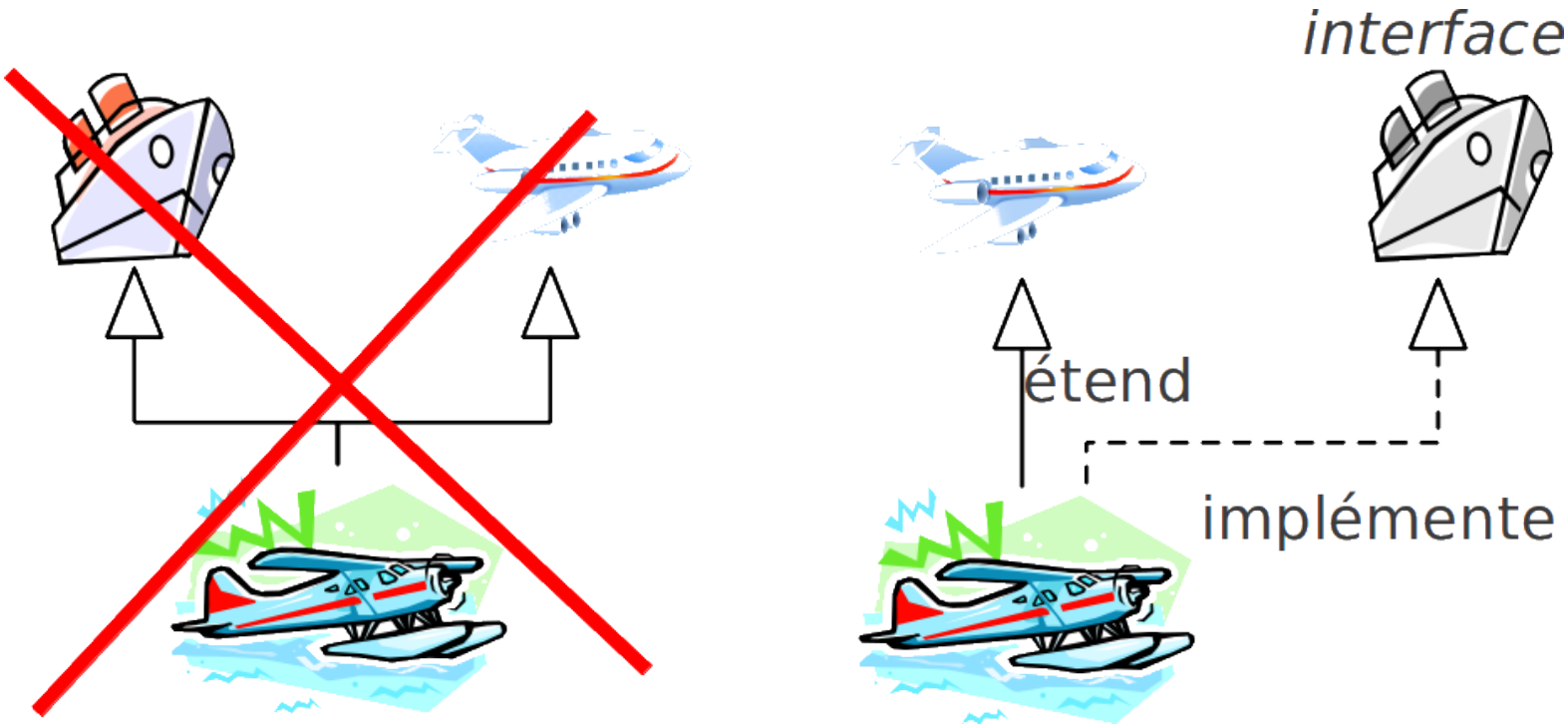
Méthodes et classes abstraites

```
public abstract class Vehicule1 {
  public void afficher() {
    System.out.println("Vehicule");
  }
}

public abstract class Vehicule2 {
  abstract public void afficher() ;
}

class Voiture2a extends Vehicule2 {
} // cette classe n'est pas instanciable

class Voiture2b extends Vehicule2 {
  public void afficher() { ... }
}
```

Interface

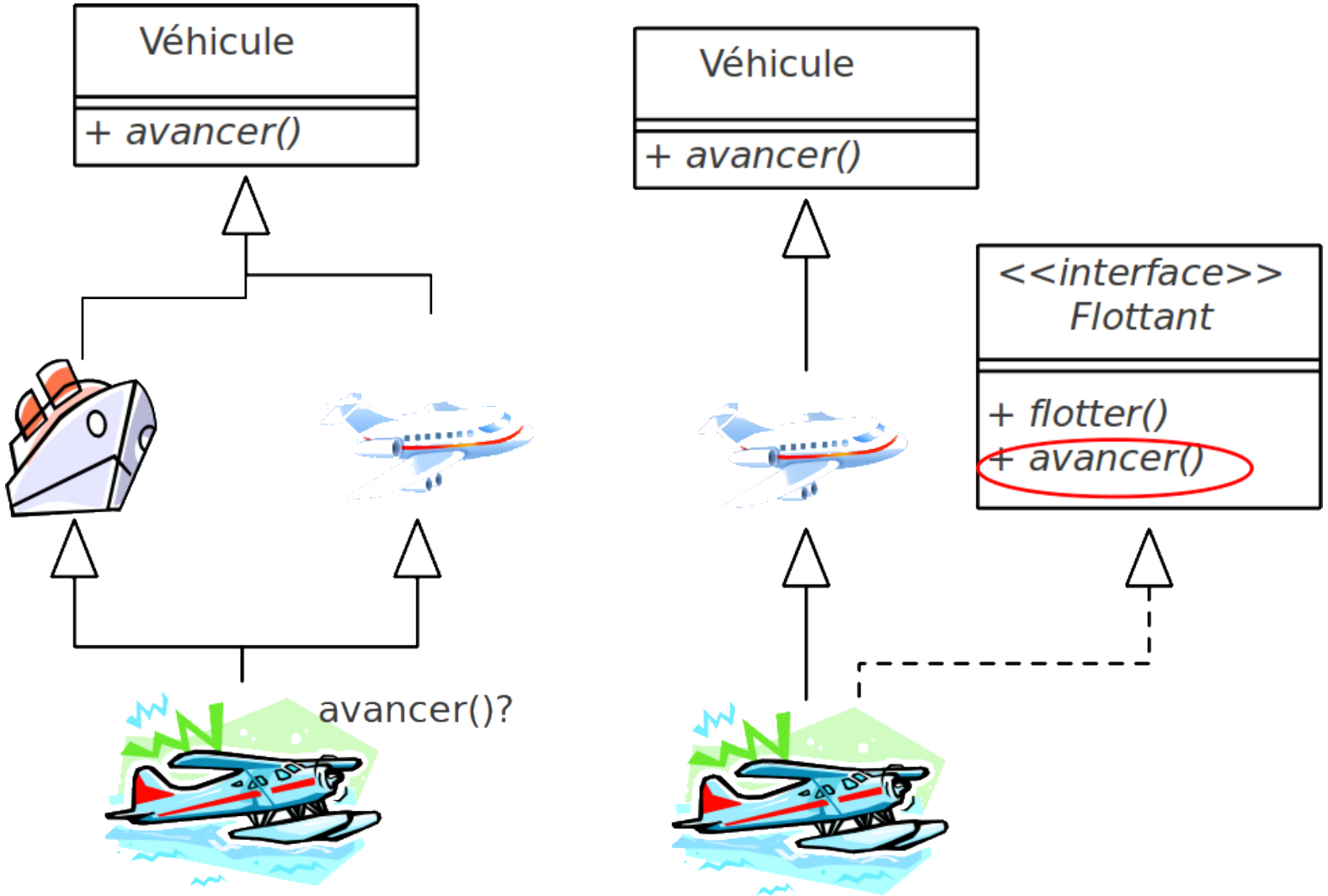
- "Classe virtuelle pure"
  - Pas de code
  - Pas de variable/attribut
  - "Constantes" autorisées (`static final`)
- Toutes les méthodes sont `abstract` par défaut
- Réponse à l'héritage multiple
- Instancier une classe ? Implémenter **toutes** les méthodes des interfaces qu'elle utilise
- Polymorphisme : la classe est du type de l'interface qu'elle implémente

Héritage multiple ?



Relation non symétrique = raison fonctionnelle

Héritage multiple ?



Interface

```
public class C extends A, B { // IMPOSSIBLE
        ...
}

interface IB {
        static final int CONSTANTE = 30;
        public int methode_de_IB (double);
}

public class C extends A implements IB {
        void methode_de_A()  {} // redefinition
        void methode_de_IB() {} // impementation
}
```

Plan

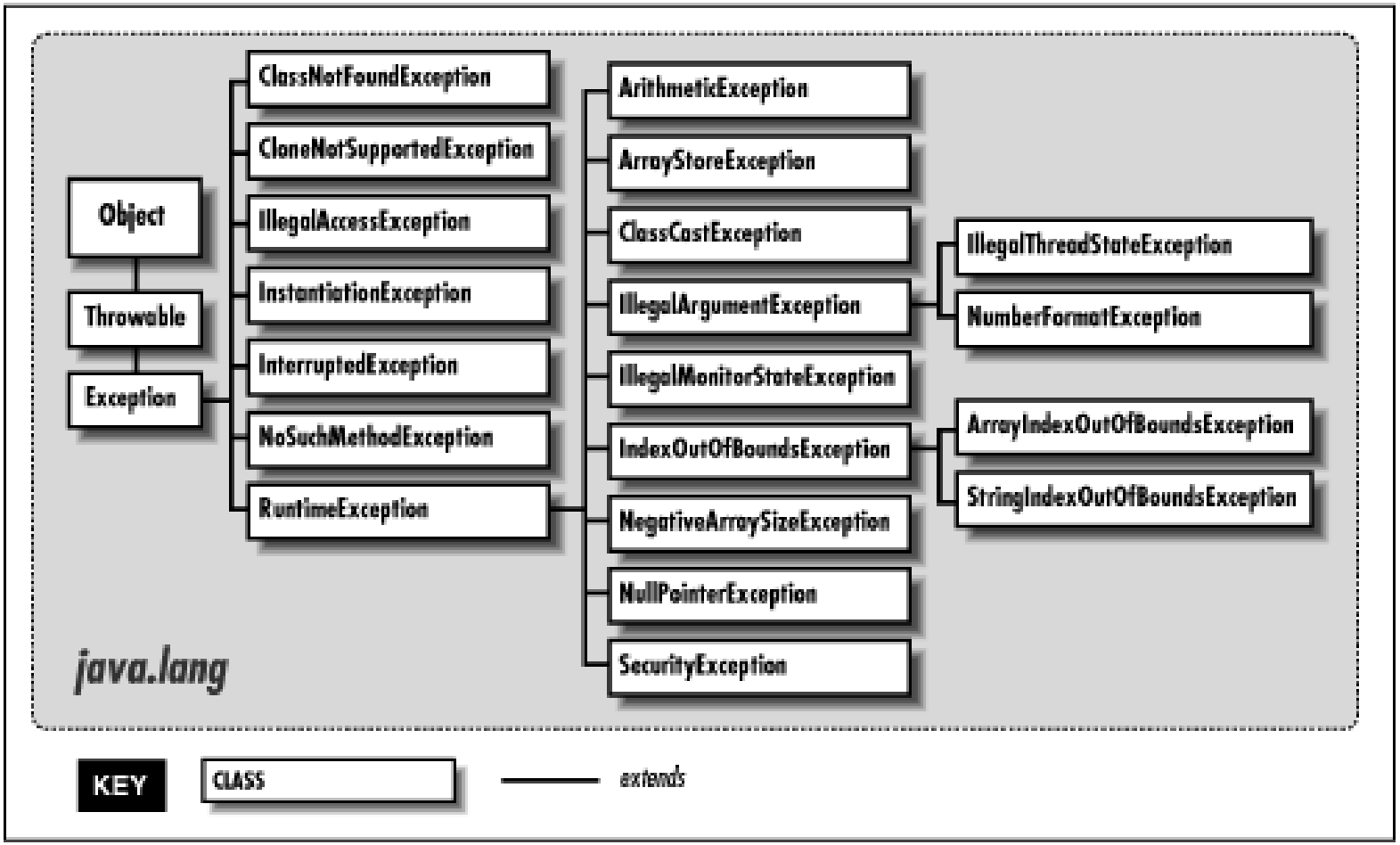
Exceptions

?

Exceptions

- Système de gestion d'erreur
  - Pendant l'execution
  - Rattrapable
- L'objet exception peut contenir
  - Message de l'erreur
  - Pile d'exécution
  - Ligne de l'erreur
- Exceptions prédéfinies

Exceptions


Object
Throwable
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
java.lang
KEY
CLASS
extends

Gérer une exception

```
try {
   readFromFile("nom_fichier");
   ...
} catch (Exception e){
   System.out.println("Exception␣lors␣de␣la␣lecture:␣" + e);
}
```

Exceptions controlée et non controlées

- Controlées
  - Héritent de Exception
  - Obligent à avoir un `try` / `catch`
  - Ex : fichier manquant, reseau non disponible...
- Non Controlées
  - Héritent de RuntimeException
  - Ex : manque de mémoire / dépassement de taille de tableau...

Exception Controlée

- throws

```
void readFromFile( String path )
                throws IOException, ParseException{
   ...
}
```

Lever une exception

- throw

```
throw new Exception("Erreur dans ma fonction traitement");

throw new SecurityException("Acces au fichier "
                            + filename
                            + " interdit");

throw new MonException();
```

Quand les utiliser ?

- Une exception allège votre code
- Une exception est coûteuse en calcul
- Ne lever des exceptions que pour les cas exceptionnels
- Ne remplace pas les code de retour (-1, null,...)
- Utiliser les `Exception` pour les cas "récupérables" et les `RuntimeException` pour les erreurs de programmation. [2]

2. Effective Java : Item 58

Gérer une exception

```
try {
   readFromFile("nom_fichier");
   ...
} catch ( Exception e ) {
   System.out.println("Exception␣levee␣"
                      + "lors␣de␣la␣lecture:␣" + e);
}
```

Plan

Collections d'objets

- Stockage de multiples objets
  - Outil fondamental de n'importe quel langage
- Tableaux
  - Taille fixe
- Collections
  - Taille variable
- Deux groupes
  - Collection
  - Map

Tableaux

```
public class Zoo {
  static final int NB_ANI = 50;
  Animal[] animaux;
  public Zoo() {
    // pas de creation d'objet, sinon le constructeur
    // par defaut serait obligatoire
    animaux = new Animal[NB_ANI];
  }
  public void placerAnimal(int i, Animal a) {
    if ((i>=0) && (i<NB_ANI))
    animaux[i] = a;
  }
  public static void main(String[] chaines) {
    Zoo zoo = new Zoo();
    zoo.placerAnimal(0, new Animal("lion"));
  }
}
class Animal {
  String nom;
  // public Animal() { nom="INCONNU"; }
  public Animal(String nom) {
    this.nom = nom;
  }
}
```

Tableaux

```java
// ajouter dans la classe Zoo
public Animal quelAnimal(int i) {
  return animaux[i];
}

// ajouter dans la class Animal
public void afficher() {
  System.out.println(nom);
}

// ajouter dans la methode main()
zoo.quelAnimal(0).afficher();  // c'est bon
// c'est la meme chose que d'ecrire
// zoo.animaux[0].nom
// si animaux et nom sont publics
zoo.quelAnimal(1).afficher();  // NullPointer
zoo.quelAnimal(60).afficher(); // ArrayOutOfBounds
```

Collection

- Interface ne définissant pas la structure

```
boolean add(Object o)
boolean remove(Object o)
boolean contains(Object o)
int size()
boolean isEmpty()
iterator iterator()
```

- java.util.Collection

Conversion collection - tableau

```
public Object[] toArray()
public Object[] toArray(Object[] a)

List liste = Arrays.asList(monTableau);

String[] tab = new String[50];
List<String> list = Arrays.asList(tab);
```

Itérateurs

- Objet qui permet de boucler sur une suite de valeurs
- Trois méthodes
  - `Object next()`
  - `boolean hasNext()`
    - true si il y a encore un element à parcourir
    - false si on est au bout
  - `void remove()`

Itérateurs

```
public void printElements(Collection c, PrintStream sortie){

  Iterator it = c.iterator();

  while (it.hasNext()) {
    sortie.println(it.next());
  }
}
```

Itérateurs suppression

```
public void printElements(Collection<String> c,
                          PrintStream sortie) {
  Iterator<String> it = c.iterator();

  while (it.hasNext()) {
    String currentString = it.next();
    if (currentString.equals("StringASupprimer")) {
      it.remove();
      sortie.println("Element supprime");
    } else {
      sortie.println(currentString);
    }
  }
}
```

Itérateurs suppression

- Pas implémenté par tous les Iterator
  - `UnsupportedOperation-Exception`
- Si on appelle `remove()` avant `next()` ou `remove()` deux fois
  - IllegalStateException

Collections – Set

- Set
  - Même méthodes que Collection
  - Duplication interdite
  - `add()` renvoie false si duplication
- SortedSet
  - Set trié

Collections – List

- List
  - Accès à des emplacements spécifiques

```
public void add(int index, Object element)
public void remove(int index)
public Object get(int index)
public Object set(int index, Object element)
```

Collections – List

- Interface
- Implémentations
  - `LinkedList` : liste chainée
  - `ArrayList` : tableau redimensionnable
  - `Vector` : tableau redimensionnable thread safe (ancienne API)
  - `Stack`

Map

- Collection de clé/valeur
- Dictionnaire
- Interface `java.util.Map`

Map

```
public Object put(Object cle, Object valeur)
public Object get(Object cle)
public Object remove(Object cle)
public int size()
public Set keySet()
public Collection values()
```

- Implémentation
  - HashMap
  - HashTable
  - LinkedHashMap
  - ...

Sorted Map

- Map triée sur ses clefs
  - `SortedMap`
  - Les éléments doivent implémenter `Comparable`

Collections – Tri

- Fonctionnalité primordiale des Collections
- Tri automatique sur les types de base
- Implémentation spécifique si besoin

Collections – Tri

```
public static void sort(List liste)

public interface Comparable{
  public int compareTo(Object o);
}

public static void sort(List liste, Comparator c)

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

Références utilisées pour le cours

- http ://fr.wikipedia.org
- http ://stackoverflow.com
- Introduction à Java (Pat Neimeyer et Jonathan Knudsen, éditions O'Reilly)
- Effective Java Second Edition (Joshua Blosh)
- Cours de Java de Loic Yon (ISIMA)