

# Bases de la programmation orientée objet

## Sous-typage et conception

Denis Poitrenaud

IUT de l'Université Paris Descartes  
Denis.Poitrenaud@ParisDescartes.fr

DUT Informatique – 2ème Semestre

## Grandes lignes de ce cours

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

# Programmer vers des interfaces

## Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

## Délégation *versus* Héritage

## Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

## Généricité et sous-typage

## Quelques interfaces de l'API standard

## Les exceptions et l'héritage

# Préférer le sous-typage à l'héritage

- ▶ Permettre de changer *a posteriori* l'implémentation
- ▶ Supprimer les contraintes d'héritage
- ▶ Abstraire des fonctions (foncteurs)
- ▶ Inverser les dépendances néfastes
- ▶ Spécialiser les services offerts en fonction des rôles des utilisateurs

```
public class Appli {
    public static void main(String[] args) {
        final int NB = 10000, NB_LECTURES = NB * 100;
        List<Integer> liste;

        liste = new LinkedList<Integer> ();
        for (int i = 0; i < NB; ++i)
            liste.add(i);

        for (int i = 0; i < NB_LECTURES; ++i)
            System.out.println(liste.get(i % NB));
    }
}
```

## Interface de l'API standard – java.util.List<E>

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

    boolean addAll(int index, Collection<? extends E> c);
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```

## Mesures de performance

```
public class Appli {
    public static void main(String[] args) {
        final int NB = 10000, NB_LECTURES = NB * 100;
        List<Integer> liste;

        liste = new LinkedList<Integer> ();
        for (int i = 0; i < NB; ++i)
            liste.add(i);

        long début = System.currentTimeMillis();

        for (int i = 0; i < NB_LECTURES; ++i)
            liste.get(i % NB);

        long temps = System.currentTimeMillis() - début;

        System.out.println((temps / 1000.) + " s secondes");
    }
}
```

Temps d'exécution : 3,5 secondes

## Modification *a posteriori*

```
public class Appli {
    public static void main(String[] args) {
        final int NB = 10000, NB_LECTURES = NB * 100;
        List<Integer> liste;

        liste = new ArrayList<Integer> ();
        for (int i = 0; i < NB; ++i)
            liste.add(i);

        long début = System.currentTimeMillis();

        for (int i = 0; i < NB_LECTURES; ++i)
            liste.get(i % NB);

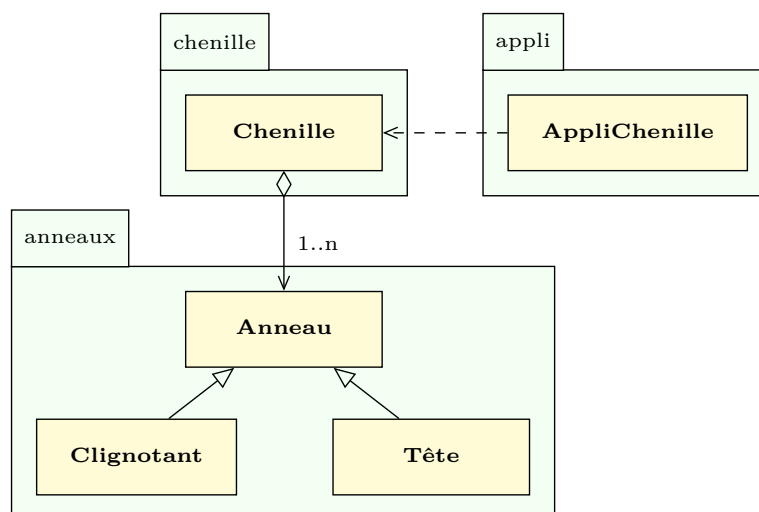
        long temps = System.currentTimeMillis() - début;

        System.out.println((temps / 1000.) + " secondes");
    }
}
```

Temps d'exécution : 0,17 secondes (soit 20 fois plus vite !!!)

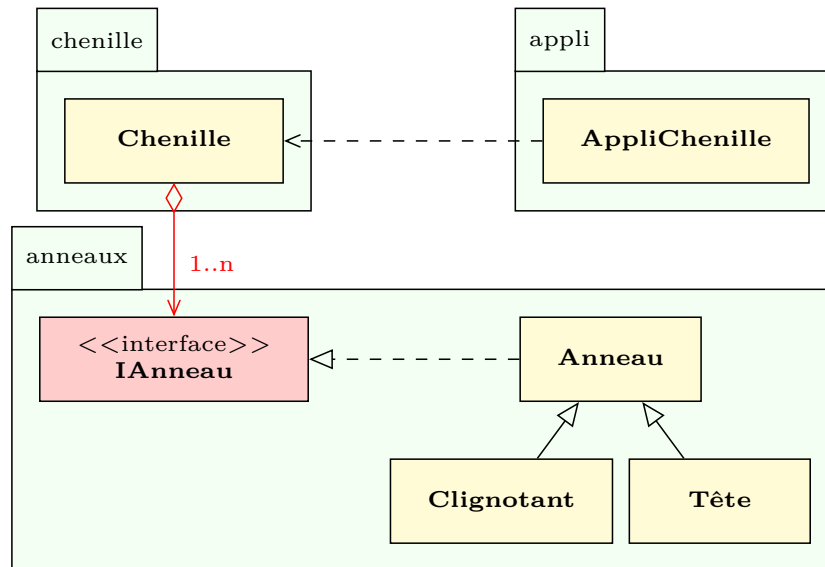
## L'héritage est une contrainte

- Intégrer un nouveau type d'anneau au sein d'une chenille passe nécessairement par une nouvelle classe héritant de la classe Anneau



- La nouvelle classe peut ne pas avoir besoin des attributs d'un anneau (héritage forcé)
- La nouvelle classe peut déjà hériter d'une classe (impossibilité  $\Rightarrow$  héritage unique)

## Une interface n'est pas une contrainte



- Intégrer un nouveau type d'anneau peut se faire aussi bien en implémentant directement l'interface IAnneau qu'en héritant de la classe Anneau (ou d'une de ses sous-classes)

## Abstraction de service – Problème

```
public class Personne {
    public enum Genre {FEMININ, MASCULIN}

    private String nom;
    private Genre genre;

    public Personne(String nom, Genre genre) {
        this.nom = nom; this.genre = genre;
    }

    public String nom() { return nom; }

    public Genre genre() { return genre; }

    public String toString() {
        String s = (genre == Genre.FEMININ ? "Mme" : "M") + " ";
        return s + nom;
    }
}
```

## Abstraction de service – Problème

Un algorithme de tri (peu efficace)

```
public class Appli {
    public static void triABulle(Personne[] t) {
        for (int i = 0; i < t.length; ++i)
            for (int j = 0; j < t.length - 1 - i; ++j)
                if (t[j].nom().compareTo(t[j + 1].nom()) > 0) {
                    Personne tmp = t[j];
                    t[j] = t[j + 1];
                    t[j + 1] = tmp;
                }
    }
}
```

Comment adapter cet algorithme à toute relation d'ordre ?

## Abstraction de service – Solution

En introduisant une interface :

```
public interface Compareur {
    boolean plusGrand(Personne a, Personne b);
}
```

```
public class Appli {
    public static void triABulle(Personne[] t,
                                Compareur c) {
        for (int i = 0; i < t.length; ++i)
            for (int j = 0; j < t.length - 1 - i; ++j)
                if (c.plusGrand(t[j], t[j + 1])) {
                    Personne tmp = t[j];
                    t[j] = t[j + 1];
                    t[j + 1] = tmp;
                }
    }
    ...
}
```

## Abstraction de service – Utilisation

```
public class OrdreGalant implements Compareteur {
    @Override
    public boolean plusGrand(Personne a, Personne b) {
        if (a.genre() == b.genre())
            return a.nom().compareTo(b.nom()) > 0;
        return a.genre() == Genre.MASCULIN;
    }
}
```

```
public class Appli {
    ...
    public static void main(String[] args) {
        Personne[] tab = {
            new Personne("Marcel", Genre.MASCULIN),
            new Personne("Louise", Genre.FEMININ),
            new Personne("Albert", Genre.MASCULIN)
        };
        triABulle(tab, new OrdreGalant());
        System.out.println(Arrays.toString(tab));
    }
} // Ce programme affiche "[Mme Louise, M Albert, M Marcel]"
```

## Abstraction de service – Utilisation avec classe anonyme

```
public class Appli {
    ...
    public static void triABulle(Personne[] t) {
        Compareteur comp = new Compareteur() {
            @Override
            public boolean plusGrand(Personne a, Personne b) {
                return a.nom().compareTo(b.nom()) > 0;
            }
        };
        triABulle(t, comp);
    }

    public static void main(String[] args) {
        Personne[] tab = {
            new Personne("Marcel", Genre.MASCULIN),
            new Personne("Louise", Genre.FEMININ),
            new Personne("Albert", Genre.MASCULIN)
        };
        triABulle(tab);
        System.out.println(Arrays.toString(tab));
    }
} // Ce programme affiche "[M Albert, Mme Louise, M Marcel]"
```

# Inversion de dépendance

## Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

## Délégation *versus* Héritage

## Création et copie d'objets

Création d'objets et polymorphisme

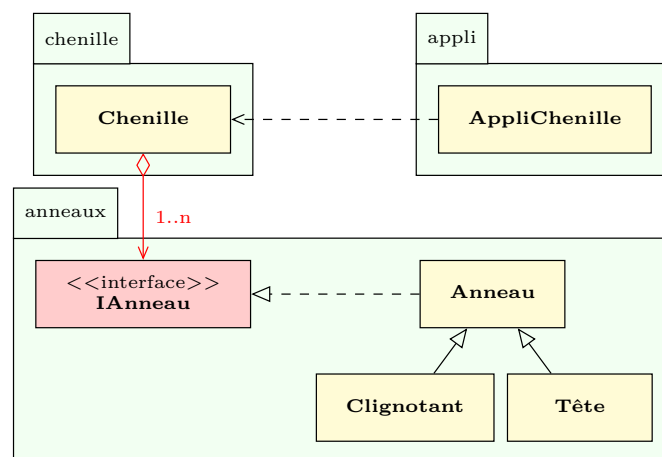
Copie d'objets

## Généricité et sous-typage

## Quelques interfaces de l'API standard

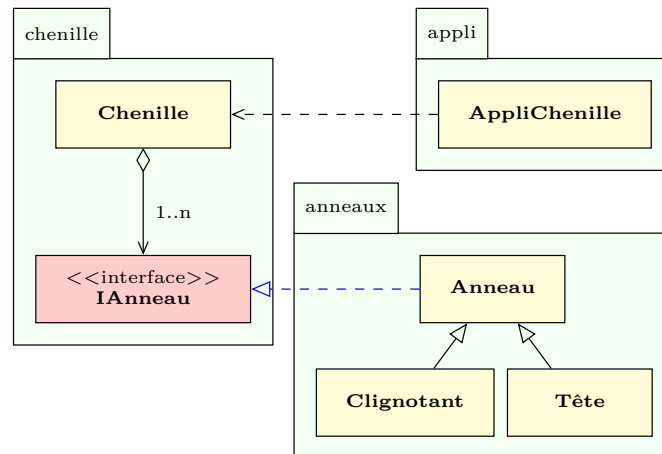
## Les exceptions et l'héritage

# Dépendance et stabilité



- ▶ La classe `Chenille` dépend de l'interface `IAnneau` ⇒ le paquetage `chenille` dépend du paquetage `anneaux`
- ▶ Le paquetage `anneaux` a vocation à être enrichi (avec des classes implémentant `IAnneau`) ⇒ le paquetage `anneaux` est instable
- ▶ La classe `Chenille` n'a pas vocation à évoluer aussi rapidement ⇒ le paquetage `chenille` est plus stable

## Inversion de dépendance



- ▶ Un paquetage ne devrait pas dépendre d'un paquetage lui étant moins stable
- ▶ Déplacer l'interface IAnneau vers le paquetage chenille suffit à inverser la dépendance
- ▶ Ce déplacement est justifié car l'interface IAnneau est de même stabilité que la classe Chenille

## Adaptation d'interface

### Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

### Délégation *versus* Héritage

### Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

### Généricité et sous-typage

### Quelques interfaces de l'API standard

### Les exceptions et l'héritage

## Adaptation d'interface – Problème

```
public class Bulletin {
    private String étudiant;
    private LinkedList<Float> notes;

    public Bulletin(String étudiant) {
        this.étudiant = étudiant;
        this.notes = new LinkedList<Float>();
    }

    public void ajouterNotes(float n) { notes.add(n); }

    public float moyenne() {
        float cumul = 0;
        for (float note : notes)
            cumul += note;
        return cumul / notes.size();
    }

    @Override
    public String toString() { return étudiant + " : " + moyenne(); }
}
```

Un étudiant consultant son bulletin ne devrait pas pouvoir ajouter des notes !

## Adaptation d'interface – Solution

En introduisant une interface (**Interface Segregation Principle**)

```
public interface BulletinEtudiant {
    float moyenne();
    String toString();
}
```

```
// visibilité paquetage
class Bulletin implements BulletinEtudiant {
    ...
    @Override
    public float moyenne() {
        ...
    }
    ...
}
```

Si le type Bulletin n'est pas visible aux étudiants et qu'il ne leur est fourni que des références de type BulletinEtudiant, ils ne pourront pas ajouter de notes

# Délégation *versus* Héritage

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

## Délégation – Principe

La délégation consiste pour une classe à déléguer les services qu'elle doit rendre (ou une partie de ceux-ci) à un autre objet

Cet autre objet est appelé *le délégué*

```
public class ImprimanteRéelle {
    public void imprimer(String document) {
        System.out.println("imprime_□" + document);
    }
}
```

```
public class Imprimante {
    // le délégué
    private ImprimanteRéelle imp = new ImprimanteRéelle();

    public void imprimer(String document) {
        imp.imprimer(document);
    }
}
```

## Délégation *versus* Héritage

### Inconvénients

- ▶ Tous les services du délégué devant être rendus visibles doivent être explicitement codés
- ▶ Dans l'exemple précédent, il n'est pas possible d'écrire du code polymorphe (une imprimante n'est pas une imprimante réelle)

### Avantages

- ▶ Il est possible de sélectionner les services du délégué rendus visibles
- ▶ Il est possible de changer dynamiquement de délégué (durant l'exécution)

Sélectionner les services rendus visibles n'est pas possible avec l'héritage. Lorsqu'une méthode est spécialisée dans une sous-classe, sa visibilité peut être augmentée (ex. `protected` → `public`) mais elle ne peut pas être réduite (ex. `public` ↯ `private`)

## Délégation – un exemple avec "mutation"

L'usage des interfaces permet de favoriser la cohérence de type (et donc le polymorphisme)

```
public interface Imprimante {  
    void imprimer(String document);  
}
```

```
public class ImprimanteHP implements Imprimante {  
    public void imprimer(String document) {  
        System.out.println("L'HP imprime " + document);  
    }  
}
```

```
public class ImprimanteEpson implements Imprimante {  
    public void imprimer(String document) {  
        System.out.println("L'Epson imprime " + document);  
    }  
}
```

## Délégation – un exemple avec "mutation" (suite)

```
public class ImprimantePerso implements Imprimante {
    // le délégué
    private Imprimante imprimante = new ImprimanteHP();

    // les mutations possibles
    void changerVersHP() {
        imprimante = new ImprimanteHP();
    }

    void changerVersEpson() {
        imprimante = new ImprimanteEpson();
    }

    // le service
    public void imprimer(String document) {
        imprimante.imprimer(document);
    }
}
```

## Délégation – un exemple avec "mutation" (suite)

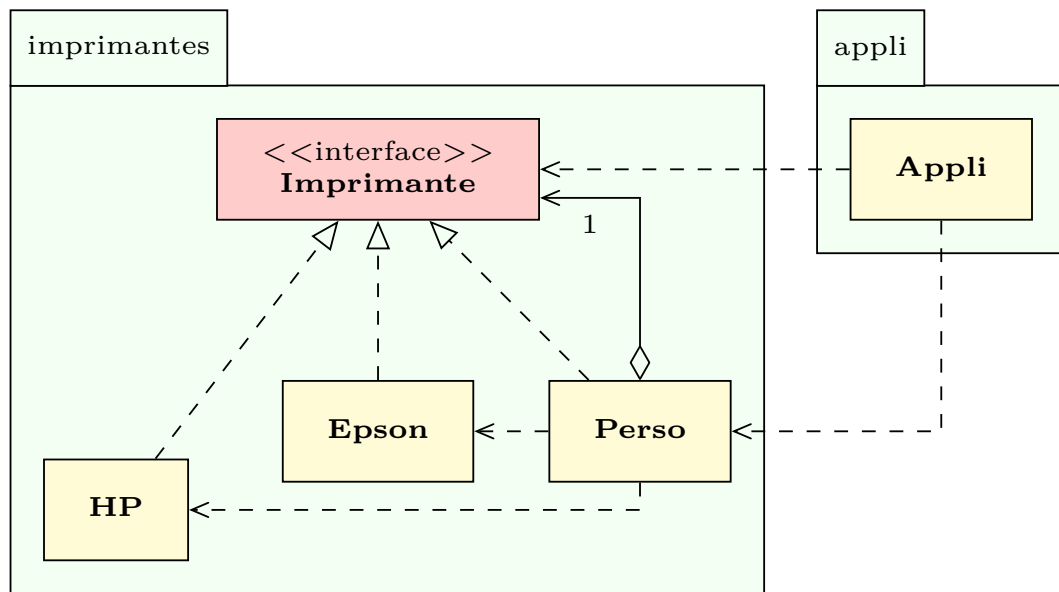
```
public class AppliImprimante {
    // Code polymorphe
    public static void impression(Imprimante imp,
                                  String doc) {
        imp.imprimer(doc);
    }

    public static void main(String[] args) {
        ImprimantePerso monImprimante = new ImprimantePerso();
        impression(monImprimante, "cv.txt");
        // mutation
        monImprimante.changerVersEpson();
        impression(monImprimante, "cv.txt");
    }
}
```

L'HP imprime cv.txt

L'Epson imprime cv.txt

## Délégation – Diagramme de classe



## Création et copie d'objets

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

# Création d'objets et polymorphisme

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

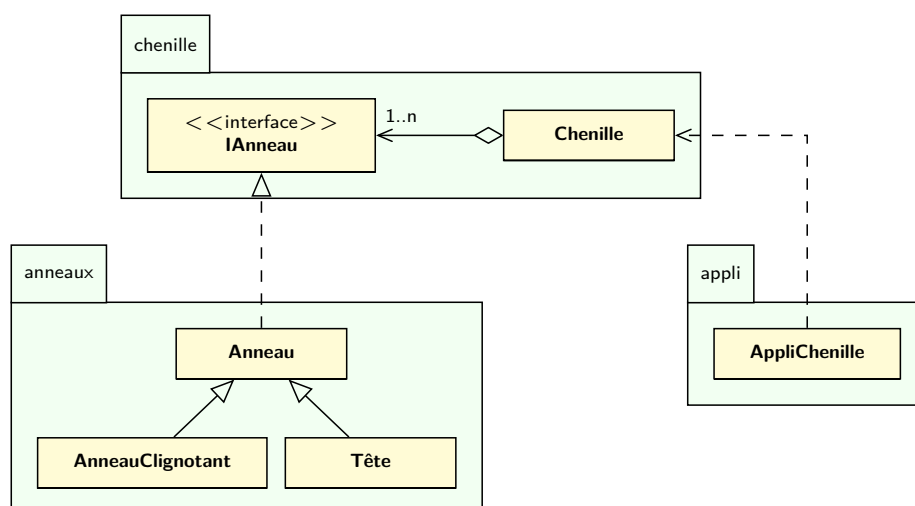
Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

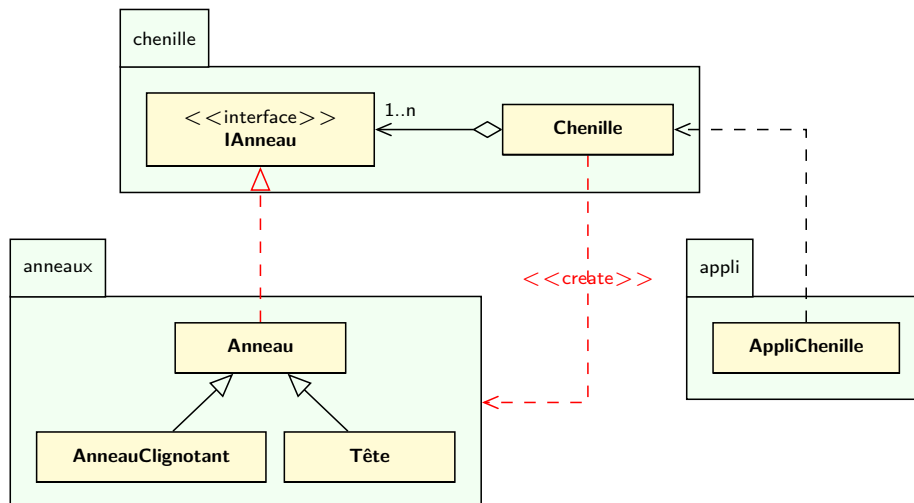
## Exemple – La chenille



- ▶ Une chenille ne dépend que de l'interface IAnneau
- ▶ Toutefois, lors de sa construction, **une chenille doit créer les anneaux** qui la composent

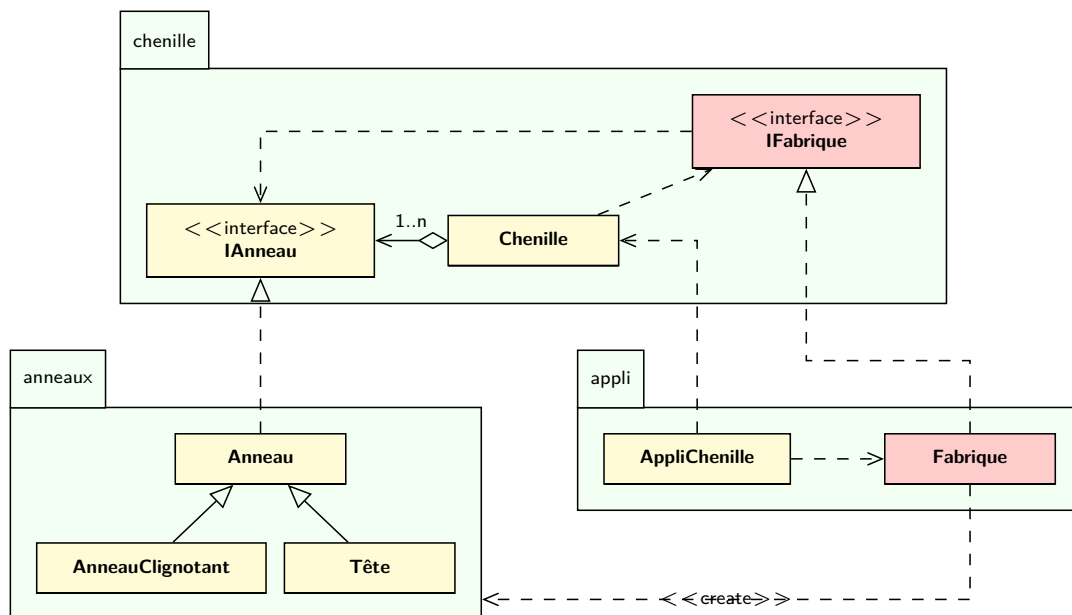
## Chenille – Dépendances circulaires

Si la chenille crée elle-même les anneaux



- ▶ Cette nouvelle dépendance crée un **circuit** entre chenille et anneaux
- ▶ Les deux paquetages sont interdépendants : il faut les fusionner ou casser le circuit

## Chenille – Introduction d'une fabrique



- ▶ Le circuit de dépendances est cassé
- ▶ L'application peut choisir le type des anneaux composant la chenille

## Chenille – paquetage chenille

```
// IFabrique.java
public interface IFabrique {
    public IAnneau getAnneau(int num, int x, int y);
}
```

```
// Chenille.java
public class Chenille {
    private IAnneau[] anneaux;

    public Chenille(int nbAnneaux, int xMax, int yMax,
                    IFabrique f) {
        anneaux = new IAnneau[nbAnneaux + 1];
        for (int i = 0; i < anneaux.length; ++i)
            anneaux[i] = f.getAnneau(i, xMax/2 - i, yMax/2);
    }
    ...
}
```

## Chenille – paquetage appli

```
// Fabrique.java
public class Fabrique implements IFabrique {
    public IAnneau getAnneau(int num, int x, int y) {
        if (num == 0) return new Tête(x, y);
        else if (num % 3 != 0) return new Anneau(x, y);
        else return new AnneauClignotant(x, y);
    }
}
```

```
// AppliChenille.java
public class AppliChenille {
    ...
    public static void main(String[] args) {
        final int NBA = 7, XMAX = 60, YMAX = 15;
        Chenille c = new Chenille(NBA, XMAX, YMAX,
                                  new Fabrique());
    }
}
```

## Chenille – une fabrique simple

- ▶ Le type concret d'anneau construit par la fabrique est déterminé uniquement en fonction du numéro de l'anneau
- ▶ La chenille crée des anneaux uniquement lors de la construction
- ▶ En conséquence, la fabrique est passée au constructeur de chenille

### Variantes possibles

- ▶ D'autres critères propres à chaque fabrique peuvent être mis en œuvre
- ▶ Si des anneaux devaient être construits a posteriori, la chenille pourrait mémoriser la fabrique et l'invoquer lorsqu'elle le souhaite

## Autre exemple – une fabrique plus complexe

Réalisation d'un éditeur graphique limité à deux types de forme : des segments et des carrés

La hiérarchie suivante a été définie

```
public interface IFigure {  
    void dessiner();  
}
```

```
public class Segment implements IFigure {  
    private Coord début, fin;  
  
    public Segment(Coord début, Coord fin) {  
        this.début = début;  
        this.fin = fin;  
    }  
  
    public void dessiner() {  
        System.out.println("segment de " + début +  
                           " à " + fin);  
    }  
}
```

## La classe Carré

```
public class Carré implements IFigure {
    private double hauteur;
    private Coord centre;

    public Carré(double hauteur, Coord centre) {
        this.hauteur = hauteur;
        this.centre = centre;
    }

    public void dessiner() {
        System.out.println("carré en " + centre +
            " de hauteur " + hauteur);
    }
}
```

## L'application principale

```
public class AppliEditeur {
    public static void main(String[] arg) {
        Editeur editeur = new Editeur();
        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.println(
                "(1)ajouter, (2)retirer ou (3)quitter");
            switch(sc.nextInt()) {
                case 1 : editeur.ajouter();
                        break;
                case 2 : editeur.retirer();
                        break;
                case 3 : return;
            }
            editeur.dessinerTout();
        }
    }
}
```

## La classe Editeur

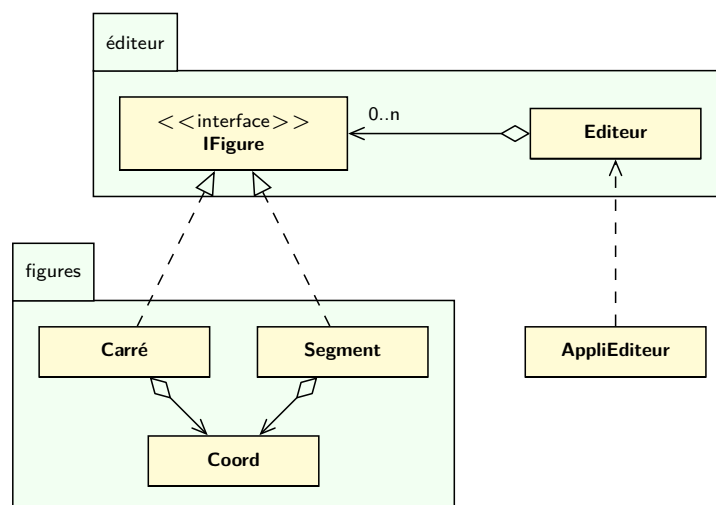
```
public class Editeur {
    private ArrayList<IFigure> figures =
        new ArrayList<IFigure>();

    public void dessinerTout() {
        for (IFigure f : figures)
            f.dessiner();
    }

    public void ajouter() {
        ...
    }

    public void retirer() {
        Scanner sc = new Scanner(System.in);
        figures.remove(sc.nextInt());
    }
}
```

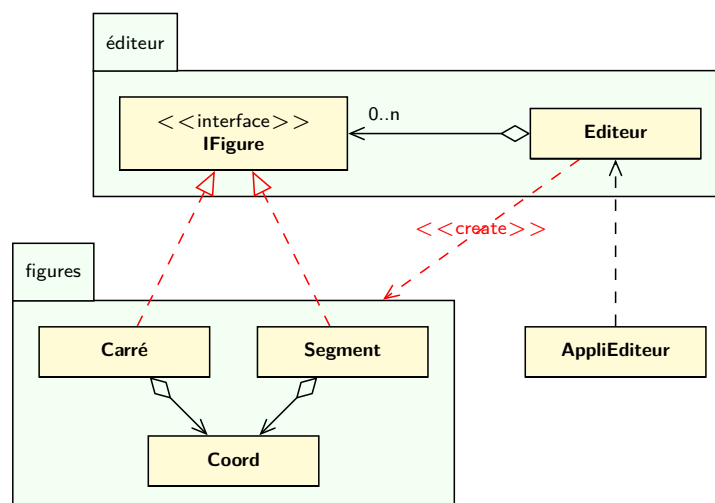
## Le diagramme de classe actuel



## La méthode Editeur.ajouter()

```
public class Editeur {
    ...
    public void ajouter() {
        IFigure f = null;
        Scanner sc = new Scanner(System.in);
        System.out.println("(1) segment ou (2) carré ");
        switch(sc.nextInt()) {
            case 1 :
                f = new Segment (
                    new Coord(sc.nextDouble(), sc.nextDouble()),
                    new Coord(sc.nextDouble(), sc.nextDouble()));
                break;
            case 2 :
                f = new Carré (sc.nextDouble(),
                    new Coord(sc.nextDouble(), sc.nextDouble()));
                break;
            default :
                System.out.println("type de figure inconnu");
        }
        if (f != null) figures.add(f);
    }
}
```

## Le diagramme de classe complet



## Problème

- ▶ La classe Editeur ne doit pas dépendre des classes concrètes de figure (en cas d'ajout de sous-classe elle devrait être modifiée)
- ▶ Il faut casser le circuit de dépendances en introduisant une fabrique
- ▶ On décide que la fabrique doit savoir construire une figure en fonction d'une chaîne de caractère

```
public interface IFabrique {  
    IFigure create (String choix);  
}
```

- ▶ Comment faire en sorte que l'éditeur puisse connaître les chaînes de caractère acceptées par la fabrique ?
- ▶ La fabrique peut fournir cette information

```
public interface IFabrique {  
    String[] getTypes ();  
    IFigure create (String choix);  
}
```

## Correction de la classe Editeur

- ▶ Dans la classe Editeur, seule la méthode ajouter a besoin de connaître la fabrique
- ▶ Un paramètre est ajouté à cette méthode

```
public class Editeur {  
    private ArrayList<IFigure> figures =  
        new ArrayList<IFigure>();  
    ...  
    public void ajouter ( IFabrique f ) {  
        String[] types = f.getTypes ();  
        int n = 0;  
        for (String type : types)  
            System.out.println(++n + " : □ " + type);  
        int choix = new Scanner (System.in).nextInt ();  
        figures.add ( f.create (types [choix - 1]));  
    }  
    ...  
}
```

## Une fabrique possible respectant ce protocole

```
public class Fabrique implements IFabrique {
    private TreeMap<String, IFigure> prototypes =
        new TreeMap<String, IFigure>();

    public void add(String s, IFigure f) {
        prototypes.put(s, f);
    }

    public String[] getTypes () {
        return prototypes.keySet().toArray(new String[0]);
    }

    public IFigure create (String choix) {
        return prototypes.get(choix).clone ();
    }
}
```

- ▶ Cette classe est indépendante des figures concrètes (elle pourrait faire partie du paquetage éditeur)
- ▶ Elle fait l'hypothèse que les figures savent **se dupliquer** (se cloner)

## Duplication des figures

```
public interface IFigure {
    void dessiner();
    IFigure clone ();
}
```

```
public class Segment implements IFigure {
    ...
    public IFigure clone () {
        return new Segment(
            new Coord(début), new Coord(fin));
    }
}
```

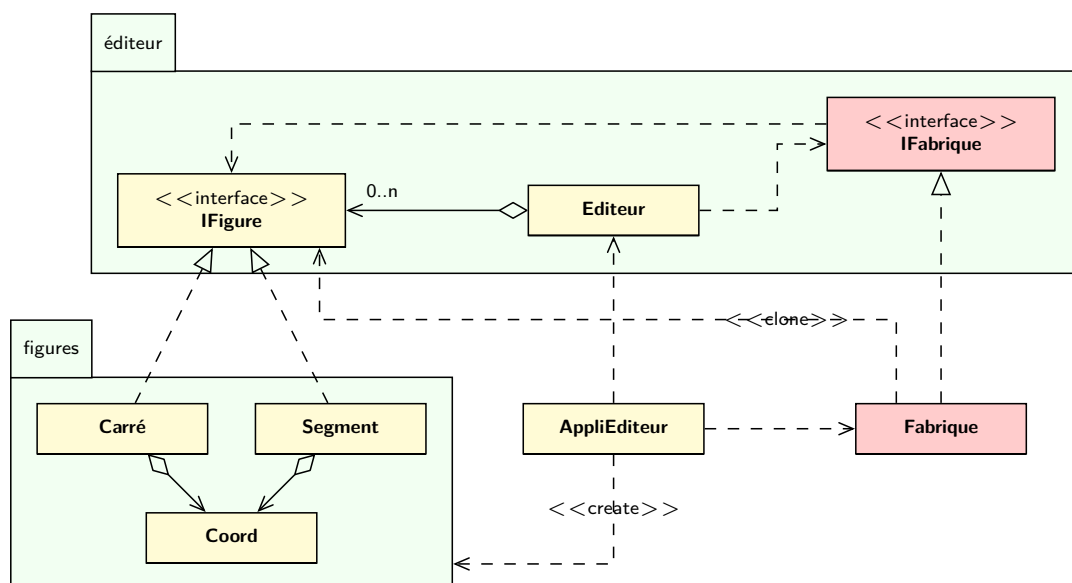
```
public class Carré implements IFigure {
    ...
    public IFigure clone () {
        return new Carré(hauteur, new Coord(centre));
    }
}
```

## L'application principale basée sur ce type de fabrique

```
public class AppliEditeur {
    public static void main (String[] arg) {
        Fabrique f = new Fabrique ();
        f.add ("Carré", new Carré(1, new Coord(0, 0)));
        f.add ("Segment",
            new Segment(new Coord(0,0), new Coord(1,1)));

        Editeur editeur = new Editeur();
        while (true) {
            System.out.println (
                "(1) ajouter, (2) retirer ou (3) quitter");
            switch (new Scanner(System.in).nextInt()) {
                case 1 : editeur.ajouter(f); break;
                case 2 : editeur.retirer(); break;
                case 3 : return;
            }
            editeur.dessinerTout();
        }
    }
}
```

## Le diagramme de classe corrigé



- ▶ Pour faciliter la maintenance, programmez vers une interface ou une classe complètement abstraite (stable), pas une implémentation (instable)
- ▶ Cachez les classes similaires derrière une interface
- ▶ Le client utilise l'interface et pas les classes concrètes
- ▶ Le client n'utilise que des méthodes abstraites (même pour la création d'instances)

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

## Copie d'objets

Si on veut modifier un objet reçu en paramètre sans impacter l'objet de l'appelant, il est nécessaire d'en faire préalablement une copie locale

- ▶ L'opérateur d'affectation (=) ne peut être appliqué à des objets car seules les références sont copiées
- ▶ Pour permettre la copie d'un objet, une classe peut
  - ▶ soit proposer une méthode particulière (ex. : `IFigure.create`)
  - ▶ soit proposer un constructeur par copie (ex. : `Coord`)
  - ▶ soit mettre en œuvre le protocole `clone`
- ▶ Selon l'implémentation, la copie peut être *superficielle* ou *profonde*
- ▶ Une copie superficielle peut être obtenue en invoquant la méthode `clone` de la classe `Object` (et en se reposant sur l'interface `Cloneable`)
- ▶ Rappel : il n'est jamais nécessaire de copier un objet immuable (`String`, `Integer`, ...)

## Constructeur par copie et protocole `clone`

**Exemple** : Les collections standard de Java proposent un constructeur par copie et implémentent le protocole `clone`

```
public class AppliCopieCollection {
    public static void main(String[] args) {
        ArrayList<Integer> liste = new ArrayList<Integer>();
        ArrayList<Integer> copie;

        for (int i = 0; i < 2; ++i) liste.add(i);

        // Obtention d'une copie via le constructeur
        copie = new ArrayList<Integer>(liste);

        // Obtention d'une copie via clone
        // (ici le transtypage "unchecked" est obligatoire)
        @SuppressWarnings("unchecked")
        copie = (ArrayList<Integer>) liste.clone();
    }
}
```

Toutefois, dans les deux cas, la **copie** reste **superficielle**

## Copie superficielle = copies distinctes avec réf. communes

```
public class Int {
    private int i;
    public Int(int i) { this.i = i; }
    public void inc() { ++i; }
    public String toString() { return Integer.toString(i); }
}

public class AppliCopieCollection {
    public static void main(String[] args) {
        ArrayList<Int> liste = new ArrayList<Int>(), copie;
        for (int i = 0; i < 2; ++i) liste.add(new Int(i));
        System.out.print("liste_=" + liste);

        copie = new ArrayList<Int>(liste);
        copie.get(0).inc();
        System.out.print(",_liste_=" + liste);
        System.out.println(",_copie_=" + copie);
    }
}
```

liste = [0, 1], liste = [1, 1], copie = [1, 1]

## Copie profonde

### Introduction d'un constructeur par copie

```
public class Int {
    ...
    public Int(Int e) {
        this(e.i);
    }
}
```

### Introduction d'une méthode de copie profonde

```
public class AppliCopieCollection {
    public static ArrayList<Int> copieProfonde(
        ArrayList<Int> ints) {
        ArrayList<Int> copie = new ArrayList<Int>();
        for (Int e : ints)
            copie.add(new Int(e));
        return copie;
    }
    ...
}
```

## Copie profonde – utilisation

```
public class AppliCopieCollection {
    ...
    public static void main(String[] args) {
        ArrayList<Int> liste = new ArrayList<Int>(), copie;
        for (int i = 0; i < 2; ++i) liste.add(new Int(i));
        System.out.println("liste_=_=" + liste);

        // Construction d'une copie via copieProfonde
        copie = copieProfonde(liste);
        copie.get(0).inc();
        System.out.println("liste_=_=" + liste);
        System.out.println("copie_=_=" + copie);
    }
}
```

liste = [0, 1], liste = [0, 1], copie = [1, 1]

## Copie superficielle ou copie profonde

- ▶ Programmer une copie profonde est généralement un exercice difficile
- ▶ Par exemple, la méthode copieProfonde vue précédemment devrait être corrigée pour supporter le cas suivant

```
public class AppliCopieCollection {
    ...
    public static void main(String[] args) {
        ArrayList<Int> liste = new ArrayList<Int>(), copie;
        Int zéro = new Int(0)
        for (int i = 0; i < 2; ++i) liste.add(zéro);
        System.out.println("liste_=_=" + liste);

        copie = copieProfonde(liste);
        liste.get(0).inc();
        copie.get(0).inc();
        System.out.println("liste_=_=" + liste);
        System.out.println("copie_=_=" + copie);
    }
}
```

liste = [0, 0], liste = [1, 1], copie = [1, 0]

## Le protocole clone

La classe `Object` dispose de la méthode suivante

```
protected Object clone()  
    throws CloneNotSupportedException
```

La méthode est protégée et donc, seules les sous-classes (i.e. toutes les classes) peuvent la spécialiser ou s'en servir (avec `super.clone()`)

Une classe autorisant la copie d'objet doit spécialiser `clone` en la rendant publique :

```
public class Int {  
    ...  
    public Int(Int e) {  
        this(e.i);  
    }  
    // la méthode peut retourner un Int (au lieu d'un Object)  
    public Int clone() {  
        return new Int(this);  
    }  
    ...  
}
```

## La méthode clone de la classe Object

Étant protégée la méthode `clone` de la classe `Object` peut être invoquée par ses sous-classes (i.e. toutes les classes).

Lorsqu'elle est invoquée, `Object.clone()`

1. Vérifie que la classe de l'appelant implémente l'interface `Cloneable` et lève l'exception `CloneNotSupportedException` si ce n'est pas le cas
2. Alloue la place mémoire nécessaire pour stocker un nouvel objet du type de l'appelant
3. Copie bit à bit les données de l'objet vers la nouvelle place mémoire (i.e. copie superficielle)
4. Retourne une référence vers ce nouvel objet

## Le protocole clone – copie superficielle basée sur `Object.clone()`

```
public class Int implements Cloneable {
    ...
    public Int(Int e) {
        this(e.i);
    }

    public Int clone () {
        try {
            return (Int)super.clone ();
        } catch (CloneNotSupportedException e) {
            // Impossible d'arriver ici
            return null;
        }
    }
    ...
}
```

## Copie d'objet – conclusion

Vaut-il mieux proposer un constructeur par copie ou implémenter le protocole `clone` ?

- ▶ Si la classe n'a pas vocation à être dérivée, les deux solutions sont équivalentes
- ▶ Dans le cas contraire, le protocole `clone` doit être privilégié (même si les avis divergent largement sur ce point)

Vaut-il mieux proposer une copie superficielle ou une copie profonde ?

- ▶ La copie superficielle est bien plus facile à développer (surtout en se reposant sur la méthode `Object.clone()`)
- ▶ Toutefois, si la copie profonde est nécessaire, l'interface `Serializable` (vue en 2ème année) peut être employée

# Généricité et sous-typage

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

# Motivation

```
class Fruit { }
class Orange extends Fruit { }
class Pomme extends Fruit { }
class Golden extends Pomme { }

public class TableauxCovariants {
    public static void main(String[] args) {
        Fruit[] fruits = new Pomme[10]; // OK - covariance

        fruits[0] = new Pomme(); // OK
        fruits[1] = new Golden(); // OK - subsumption

        fruits[0] = new Fruit(); // ArrayStoreException
        fruits[1] = new Orange(); // ArrayStoreException
    }
}
```

Ce programme compile sans erreur mais à l'exécution, le type de `fruits` est `Pomme[]` et non pas `Fruit[]` ou `Orange[]`  $\Rightarrow$  les deux dernières affectations lèvent une exception `ArrayStoreException`

## Motivation – suite

```
// Boîte où ranger des objets
public class Box {

    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

## Motivation – suite

```
public class BoxDemo {
    public static void main(String[] args) {
        Box integerBox = new Box();
        // Consigne : n'y ranger que des entiers
        integerBox.add((Integer) 10);
        Integer someInteger = (Integer) integerBox.get();
        System.out.println(someInteger);

        integerBox.add(10); // auto-boxing
        int someInt = (Integer) integerBox.get();
        System.out.println(someInt);

        integerBox.add("10"); // erreur logique
        // l'erreur ne sera vue qu'à l'exécution
        // ClassCastException
        someInt = (Integer) integerBox.get();
        System.out.println(someInt);
    }
}
```

## Problème et solution

Problème : Ces erreurs devraient être détectées à la compilation

Solution : Employer des génériques

```
import java.util.ArrayList;

class Fruit { }
class Orange extends Fruit { }
class Pomme extends Fruit { }
class Golden extends Pomme { }

class CollectionCovariantes {
    public static void main(String[] args) {
        ArrayList<Pomme> fruits = new ArrayList<Pomme>();
        fruits.add(new Pomme());
        fruits.add(new Golden()); // OK - subsumption

        fruits.set(0, new Fruit()); // erreur de compilation
        fruits.set(1, new Orange()); // erreur de compilation
    }
}
```

## Écrire une classe générique

```
public class GBox<T> {
    private T t; // T pour "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

## Utilisation

```
public class GBoxDemo {
    public static void main(String[] args) {
        GBox<Integer> integerBox = new GBox<Integer>();

        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // no cast!
        System.out.println(someInteger);

        integerBox.add(10); // auto-boxing
        int someInt = integerBox.get(); // auto-unboxing
        System.out.println(someInt);

        integerBox.add("10"); // erreur de compilation
        ...
    }
}
```

## Utilisation – suite

Les paramètres génériques (tels que T dans GBox<T>) ne peuvent être que des types 'objet'

```
public class GBoxDemo {
    public static void main(String[] args) {
        // Les types primitifs ne peuvent pas être employés
        GBox<int> intBox = new GBox<int>(); // erreur de compil.

        // mais les tableaux peuvent l'être
        GBox<int []> intsBox = new GBox<int []>(); // OK
        intsBox.add(new int [5]);
        System.out.println(Arrays.toString(intsBox.get()));
    }
}
```

## Interface générique

```
// Définie dans java.lang
public interface Comparable<T> {
    int compareTo(T o);
}
```

```
public class Int implements Comparable<Int> {
    private int x;
    public Int(int x) { this.x = x; }
    public String toString() {
        return Integer.toString(x);
    }
    public int compareTo(Int o) {
        return x - o.x;
    }
}
```

## Méthode générique

Les paramètres génériques doivent être indiqués avant le type de retour de la méthode

```
class Util {
    public static <T> void fillBoxes(T u,
                                     List<GBox<T>> boxes) {
        for (GBox<T> box : boxes) {
            box.add(u);
        }
    }
}
```

## Contrainte d'héritage

Des contraintes peuvent être imposées aux paramètres génériques

```
class Util {
    ...
    public static <T1 extends Number, T2 extends Number>
        float sommeRéelle(T1 n1, T2 n2) {
            return n1.floatValue() + n2.floatValue();
        }
}
```

Les types T1 et T2 ne peuvent être quelconques. Ils doivent étendre la classe Number

## Contrainte d'implémentation

Le type T doit implémenter l'interface Comparable<T>

```
class Util {
    ...
    public static <T extends Comparable<T>>
        void triBulle(T[] tab) {
            for (int i = 0; i < tab.length; ++i)
                for (int j = 0; j < tab.length - i - 1; ++j)
                    if (tab[j].compareTo(tab[j + 1]) > 0) {
                        T tmp = tab[j];
                        tab[j] = tab[j + 1];
                        tab[j + 1] = tmp;
                    }
        }
}
```

Un paramètre générique peut avoir une contrainte d'héritage (obligatoirement la première) et plusieurs contraintes d'implémentation (séparées par des &)

Exemple : <T extends Number & Comparable<T>>

## Généricité et covariance

```
class Util {
    ...
    public static void aMethod(GBox<Number> box) {
        System.out.println(box.get().floatValue());
        box.add(new Double(5)); // OK
    }
}
```

```
public class GBoxDemo {
    public static void main(String[] args) {
        GBox<Integer> integerBox = new GBox<Integer>();
        integerBox.add(new Integer(10));

        Util.aMethod(integerBox); // erreur de compilation
    }
}
```

Même si Integer est une sous-classe de Number, GBox<Integer> n'est pas une sous-classe de GBox<Number>

## Généricité et covariance – suite

```
class Util {
    public static void
        anotherMethod(GBox<? extends Number> box) {
        System.out.println(box.get().floatValue()); // OK
        box.add(new Integer(5)); // erreur de compilation
    }
}
```

```
public class GBoxDemo {
    public static void main(String[] args) {
        GBox<Integer> integerBox = new GBox<Integer>();
        integerBox.add(new Integer(10));

        Util.anotherMethod(integerBox); // OK
    }
}
```

- ▶ <? extends Number> doit être lu "n'importe quel type étendant la classe Number"
- ▶ Comme le type réel n'est pas connu, il n'est pas possible d'ajouter un élément dans la boîte (box est peut être de type Box<Float>)

## Quelques interfaces de l'API standard

Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

Délégation *versus* Héritage

Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

Généricité et sous-typage

Quelques interfaces de l'API standard

Les exceptions et l'héritage

## Les interfaces de l'API standard

- ▶ La bibliothèque standard de Java 7 est composée de :
  - ▶ plus de 3000 classes et
  - ▶ près de 2000 interfaces
- ▶ Les interfaces les plus fréquemment employées sont
  - ▶ `Collection<E>`, `List<E>`, `Deque<E>`, `Queue<E>`, `Set<E>`, `SortedSet<E>`
  - ▶ `Map<K, V>`
  - ▶ `Comparable<T>`, `Comparator<T>`
  - ▶ `Observer`
  - ▶ `Iterable<E>`, `Iterator<E>`

## Les interfaces Iterable<E> et Iterator<E>

```
/** Implementing this interface allows an object to be the
 * target of the "foreach" statement.
 *
 * Known Subinterfaces: Collection<E>, List<E>, ... */
public interface Iterable<E> {
    /** Returns an iterator over a set of elements of
     * type E. */
    Iterator<E> iterator();
}
```

```
/** An iterator over a collection. */
public interface Iterator<E> {
    /** Returns true if the iteration has more elements. */
    boolean hasNext();
    /** Returns the next element in the iteration. */
    E next();
    /** Removes from the underlying collection the last
     * element returned by this iterator (optional
     * operation). */
    void remove();
}
```

## Utilisation avec les listes

```
import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Itération {
    public static void main(String[] args) {
        List<Integer> liste = new LinkedList<Integer>(
            Arrays.asList(0, 9, 0, 3, 0, 7, 0));

        System.out.println(liste);
        // [0, 9, 0, 3, 0, 7, 0]
        Iterator<Integer> it = liste.iterator();
        while (it.hasNext()) {
            int val = it.next();
            if (val == 0)
                it.remove();
        }
        System.out.println(liste);
        // [9, 3, 7]
    }
}
```

## Exemple d'implémentation

On veut pouvoir parcourir facilement les trois composantes RGB d'une couleur

```
public class RGB {
    private int red, green, blue;

    public RGB(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
}
```

Une classe implémentant `Iterable<E>` peut être employée dans les boucles *foreach*

Il suffit de faire en sorte que RGB implémente `Iterable<Integer>`

## Une classe interne statique pour itérer sur un RGB

```
import java.util.Iterator;
public class RGB {
    ...
    private static class Itérateur implements Iterator<Integer> {
        private RGB rgb;
        private int color = 0;

        public Itérateur(RGB r) { rgb = r; }

        @Override
        public boolean hasNext() { return color < 3; }

        @Override
        public Integer next() {
            assert (hasNext());
            ++color;
            return color==1 ? rgb.red : (color==2 ? rgb.green : rgb.blue);
        }

        @Override // non nécessaire avec Java 8
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

## Implémentation de Iterable<Integer> et utilisation

```
public class RGB implements Iterable<Integer> {
    ...
    @Override
    public Iterator<Integer> iterator() {
        return new Itérateur(this);
    }
}
```

```
public class Appli {
    public static void main(String[] args) {
        RGB rgb = new RGB(3, 9, 2);

        for (int c : rgb)
            System.out.print(c + " ");

        // 3 9 2
    }
}
```

## Solution alternative basée sur une classe interne anonyme

```
public class RGB implements Iterable<Integer> {
    private int red, green, blue;

    public RGB(int red, int green, int blue) {
        this.red = red; this.green = green; this.blue = blue;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int color = 0;

            @Override
            public boolean hasNext() { return color < 3; }

            @Override
            public Integer next() {
                assert (hasNext());
                ++color;
                return color == 1 ? red : (color == 2 ? green : blue);
            }

            @Override // non nécessaire avec Java 8
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

## Second exemple

```
public class Matrice {
    private int[][] tab;

    public Matrice(int nbl, int nbc) {
        tab = new int[nbl][nbc];
        for (int[] ligne : tab)
            Arrays.fill(ligne, 0);
    }

    public int nbLignes() {
        return tab.length;
    }

    public int nbColonnes() {
        return tab[0].length;
    }

    public int get(int nl, int nc) {
        return tab[nl][nc];
    }

    public void set(int nl, int nc, int val) {
        tab[nl][nc] = val;
    }
}
```

## Itération

- ▶ Les matrices manipulées comportent un grand nombre de valeurs nulles
- ▶ On veut pouvoir parcourir facilement les valeurs non-nulles et connaître leur position dans la matrice
- ▶ Nous devons parcourir des triplets (valeur, ligne, colonne)

## Triplet (valeur, ligne, colonne)

```
public class Valeur {
    private int valeur, ligne, colonne;

    public Valeur(int val, int nl, int nc) {
        valeur = val;
        ligne = nl; colonne = nc;
    }

    public int valeur() { return valeur; }

    public int ligne() { return ligne; }

    public int colonne() { return colonne; }

    @Override
    public String toString() {
        return "(" + ligne + ", " + colonne + ") -> " + valeur;
    }
}
```

## Implémentation de `Iterator<Valeur>` pour une matrice

```
class ItérateurMatrice implements Iterator<Valeur> {
    protected Matrice m;
    protected int nl, nc;
    // Si le parcours est fini alors nc >= m.nbColonnes(),
    // sinon nl et nc désigne la prochaine valeur de m à retourner
    public ItérateurMatrice(Matrice m) {
        this.m = m;
        nl = 0; nc = -1;
        avance();
    }

    protected void avance() {
        do {
            ++nc;
            if (nc >= m.nbColonnes() && nl < m.nbLignes() - 1) {
                nc = 0;
                ++nl;
            }
        } while(nc < m.nbColonnes() && m.get(nl, nc) == 0);
    }

    @Override
    public boolean hasNext() {
        return nc < m.nbColonnes();
    }
    ...
}
```

## Implémentation de Iterator<Valeur> pour une matrice

```
class ItérateurMatrice implements Iterator<Valeur> {
    ...
    @Override
    public Valeur next() {
        assert(hasNext());
        Valeur val = new Valeur(m.get(nl, nc), nl, nc);
        avance();
        return val;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

```
public class Matrice implements Iterable<Valeur> {
    ...
    @Override
    public Iterator<Valeur> iterator() {
        return new ItérateurMatrice(this);
    }
}
```

## Itérateur avec suppression

- ▶ L'opération remove doit supprimer la dernière valeur retournée par next
- ▶ Dans notre cas, la suppression peut être assimilée à mettre à zéro la case correspondante de la matrice

```
public class Matrice implements Iterable<Valeur> {
    ...
    public Iterator<Valeur> iteratorWithRemove() {
        return new Itérateur (this) { // sous-classe anonyme !!
            private int pnl, pnc; // dernière position retournée

            @Override
            protected void avance() {
                pnl = nl; pnc = nc; // (nl et nc 'protected' dans Itérateur)
                super.avance();
            }

            @Override
            public void remove() {
                assert (pnc >= 0);
                m.set(pnl, pnc, 0); // (m 'protected' dans Itérateur)
            }
        };
    }
}
```

```
public class Appli {
    public static void main(String[] args) {
        Matrice m = new Matrice(2, 3);
        m.set(0, 1, 3); m.set(1, 0, 1); m.set(1, 2, 2);

        for (Valeur v : m)
            System.out.print(v + "␣/␣");
        System.out.println();

        // (0, 1) -> 3 / (1, 0) -> 1 / (1, 2) -> 2 /

        Iterator<Valeur> it = m.iteratorWithRemove();
        while (it.hasNext()) {
            Valeur v = it.next();
            if (v.valeur() == 1)
                it.remove();
        }

        for (Valeur v : m)
            System.out.print(v + "␣/␣");
        System.out.println();

        // (0, 1) -> 3 / (1, 2) -> 2 /
    }
}
```

## Les exceptions et l'héritage

### Programmer vers des interfaces

Inversion de dépendance

Adaptation d'interface

### Délégation *versus* Héritage

### Création et copie d'objets

Création d'objets et polymorphisme

Copie d'objets

### Généricité et sous-typage

### Quelques interfaces de l'API standard

### Les exceptions et l'héritage

## Erreurs de programmation – unchecked exceptions

- ▶ Les **erreurs de programmation** détectées à l'exécution sont signalées au programme par des levées d'exception
- ▶ À chaque type d'erreur correspond une classe d'exception
  - ▶ `ArithmeticException` (division par zéro)
  - ▶ `ClassCastException` (conversion vers un type auquel n'appartient pas l'objet)
  - ▶ `IndexOutOfBoundsException` (indice hors des bornes du tableau)
  - ▶ `NegativeArraySizeException` (taille négative de tableau)
  - ▶ `NoSuchElementException` (lecture erronée d'un Scanner)
  - ▶ `NullPointerException` (accès à une référence nulle)
  - ▶ `IllegalArgumentException` (paramètre ayant une valeur illégale)
  - ▶ `IllegalStateException` (objet dans un état illégal)
  - ▶ `UnsupportedOperationException` (opération non supportée par l'objet)
- ▶ Toutes ces classes sont des sous-classes de **`RuntimeException`**
- ▶ Ces exceptions ne sont généralement pas attrapées et traitées par le programme
- ▶ Elles ont vocation à signaler au programmeur que le programme contient une erreur (qui doit être corrigée !!!)

## Erreurs graves – unchecked exceptions

- ▶ Les **erreurs graves** détectées à l'exécution sont signalées au programme par des levées d'exception
- ▶ À chaque type d'erreur correspond une classe d'exception
  - ▶ `OutOfMemoryError` (demande d'allocation mémoire non satisfaite)
  - ▶ `StackOverflowError` (pile d'exécution pleine)
  - ▶ `IOException` (erreur grave d'entrée/sortie)
  - ▶ `InternalError` (erreur interne grave de la JVM)
- ▶ Toutes ces classes sont des sous-classes de **`Error`**
- ▶ Ces exceptions ne sont généralement pas attrapées et traitées par le programme
- ▶ Elles ont vocation à signaler à l'utilisateur que le programme ne peut être exécuté dans ces conditions

## Erreurs contextuelles – checked exceptions

- ▶ Des **situations particulières** détectées à l'exécution peuvent être signalées au programme par des levées d'exception
- ▶ À chaque situation correspond une classe d'exception
  - ▶ CloneNotSupportedException (protocole clone)
  - ▶ InterruptedException (interruption d'un thread)
  - ▶ EOFException (fin de fichier)
  - ▶ FileNotFoundException (fichier inconnu)
  - ▶ InterruptedIOException (entrée/sortie interrompue)
  - ▶ SocketException (problème de connexion réseau)
  - ▶ ...
- ▶ Toutes ces classes sont des sous-classes de **Exception**
- ▶ **Ces exceptions doivent être explicitement prises en compte** par le programme
- ▶ Elles peuvent être traitées là où elles sont détectées ou signalées à l'appelant

## Exemple : traitement sur place

```
public class Appli {
    private final static Scanner sc = new Scanner(System.in);

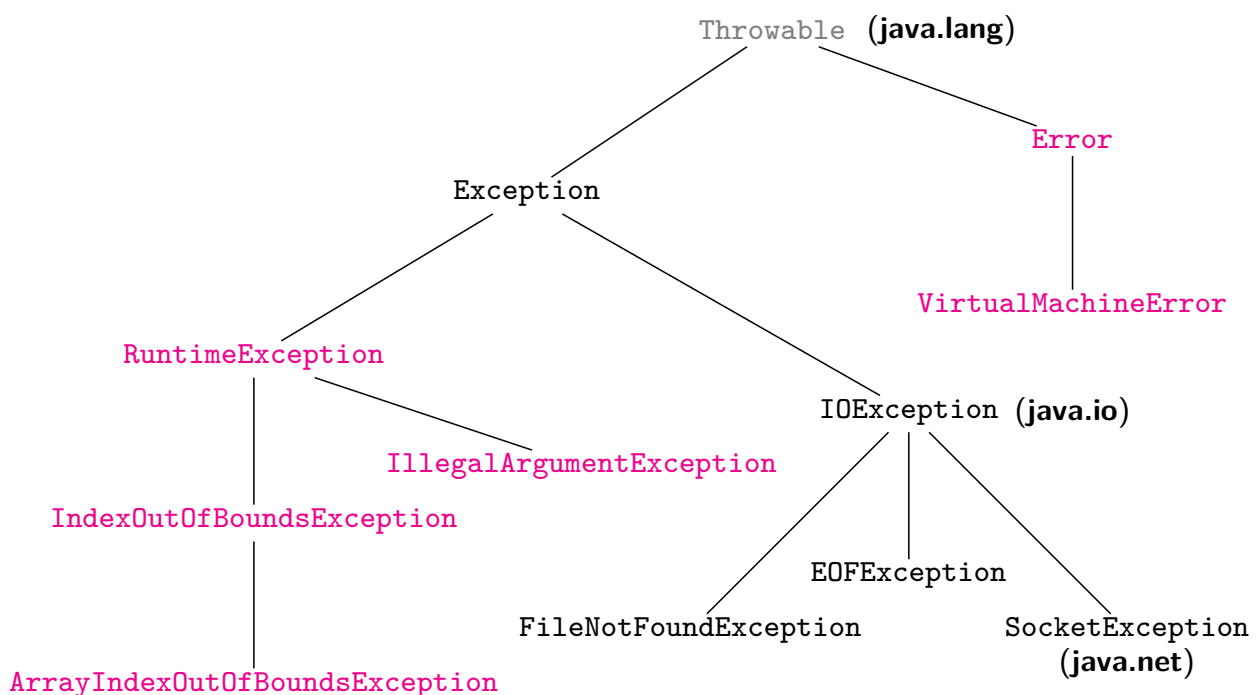
    public static void main(String[] args) {
        FileInputStream in = null;
        // ouverture contrôlée d'un fichier
        while (in == null) {
            System.out.print("entrez le nom du fichier : ");
            String nom = sc.next();
            try {
                in = new FileInputStream(nom);
            }
            catch (FileNotFoundException e) {
                System.out.println("Ce fichier n'existe pas");
            }
        }
        // lecture du fichier et affichage du contenu
        Scanner scin = new Scanner(in);
        while (scin.hasNextLine())
            System.out.println(scin.nextLine());
        scin.close(); // ferme le fichier associé (in)
    }
}
```

## Exemple : signalement à l'appelant

```
public class Appli {
    private final static Scanner sc = new Scanner(System.in);
    // la méthode signale l'erreur en laissant passer l'exception
    // levée par le constructeur de FileInputStream
    public static void affiche(String n)
        throws FileNotFoundException {
        Scanner scin = new Scanner(new FileInputStream(n));
        while (scin.hasNextLine())
            System.out.println(scin.nextLine());
        scin.close();
    }

    public static void main(String[] args) {
        boolean ok = false;
        while (!ok) {
            System.out.print("entrez le nom du fichier : ");
            String nom = sc.next();
            try {
                affiche(nom); // elle est traitée ici
                ok = true;
            }
            catch (FileNotFoundException e) {
                System.out.println("ce fichier n'existe pas");
            }
        }
    }
}
```

## Extrait de la hiérarchie standard des classes d'exception



## Utilisation du mécanisme d'exception

- ▶ Il est **obligatoire** de prendre en compte les *checked exceptions*
- ▶ Dans la bibliothèque standard cela concerne en particulier :
  - ▶ Méthode `clone` de la classe `Object`
  - ▶ Entrées/sorties (fichier, réseau, sérialisation, ...)
  - ▶ Interruptions (pour les threads)
- ▶ Il est possible de prendre en compte des *unchecked exceptions*
  - ▶ Erreurs de format dans les lecture de donnée, de conversion, etc.
- ▶ Il est possible de lever explicitement des *unchecked exceptions*
  - ▶ Préconditions (`IllegalArgumentException`, `IllegalStateException`)
- ▶ Il est possible de définir ses propres classes d'exception (*checked* ou pas)
- ▶ C'est une structure de contrôle puissante

## Illustration

```
public class Abandon {
    // une suite de traitements (composés de sous-traitements)
    public static void main(String[] args) {
        traitement1();
        traitement2();
    }

    private static void traitement1() {
        ...
    }
    private static void traitement2() {
        sousTraitement1();
        sousTraitement2();
    }

    private static void sousTraitement1() {
        ...
    }
    private static void sousTraitement2() {
        ...
    }
}
```

## Faire des sauts dans la pile d'exécution

```
public class Abandon {
    // un traitement peut décider d'abandonner sa tâche
    private static void traitement1() {
        if (condition())
            abandonner("traitement1");
    }
    private static void sousTraitement1() {
        if (condition())
            abandonner("sous-traitement1");
    }
    ...
    // signalement d'abandon par levée d'exception
    private static void abandonner(String message) {
        throw new RuntimeException(message);
    }
    // simulation aléatoire
    private static final int TAUX_REUSSITE = 30;
    private static final Random rd = new Random();
    private static boolean condition() {
        return rd.nextInt(100) >= TAUX_REUSSITE;
    }
    ...
}
```

## Faire des sauts dans la pile d'exécution

```
public class Abandon {
    ...
    // un sous-traitement peut même être récursif
    private static void sousTraitement2(int n) {
        if (n > 0)
            sousTraitement2(n - 1);
        if (condition())
            abandonner("sous-traitement2-" + n);
    }

    private static void traitement2() {
        sousTraitement1();
        sousTraitement2(10);
    }
    ...
}
```

## Faire des sauts dans la pile d'exécution

```
public class Abandon {
    ...
    // le programme est adapté pour relancer la chaîne
    // de traitement en cas d'abandon (d'où qu'il survienne)
    public static void main(String[] args) {
        boolean ok = false;
        while (!ok) {
            try {
                traitement1();
                traitement2();
                // seul le succès mène ici
                ok = true;
            }
            catch (RuntimeException e) {
                // tous les abandons mènent ici
                System.out.println("on recommence tout à cause du "
                    + e.getMessage());
            }
        }
    }
}
```

## Définir ses propres classes d'exception

### Intérêts

- ▶ Transporter de l'information lors de la remontée de l'exception
- ▶ Classifier les différentes exceptions au sein d'une hiérarchie pour faciliter leur traitement

### Implémentation

- ▶ Définir une sous-classe de `RuntimeException` (*unchecked exception*)
- ▶ Définir une sous-classe de `Exception` (*checked exception*)

### Exemple

```
public class CoupIllégal extends Exception {
    // pour la sérialisation
    private static final long serialVersionUID = 1L;

    public CoupIllégal() {
        super("coup illégal"); // message associé
    }
}
```

## Remonter de l'information

Une classe d'exception peut véhiculer de l'information

```
public class CoupIllégal extends Exception {
    private static final long serialVersionUID = 1L;

    private int x, y; // Coordonnée du coup

    public CoupIllégal(int x, int y) {
        super("coup illégal en (" + x + ", " + y + ")");
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

## Remonter de l'information

```
public class Dame {
    private static final int TAILLE = 10;
    public enum Couleur {BLANC, NOIR};

    private Couleur courant = Couleur.BLANC;
    ...
    public Couleur joueurCourant() {
        return courant;
    }

    public void jouer() throws CoupIllégal {
        int x, y;
        ...
        poser(x, y, courant);
        ...
    }

    private void poser(int x, int y, Couleur c)
        throws CoupIllégal {
        if (0 < x || x >= TAILLE || 0 < y || y >= TAILLE)
            throw new CoupIllégal(x, y); // transmission
        ...
    }
}
```

## Remonter de l'information

Celui qui traite l'erreur a accès à l'information

```
public class Appli {
    public static void main(String[] args) {
        Dame dame = new Dame();

        while (!dame.finie()) {
            try {
                dame.jouer();
            }
            catch (CoupIllégal e) { // réception
                System.out.println(dame.joueurCourant() +
                    " ne peut pas jouer en (" +
                    e.getX() + ", " + e.getY() + ")");
            }
        }
    }
}
```

## Classifier des exceptions

Une hiérarchie de classes d'exception

```
public class CoordonnéeIllégale extends CoupIllégal {
    private static final long serialVersionUID = 1L;

    public CoordonnéeIllégale(int x, int y) {
        super(x, y);
    }
}
```

```
public class CaseOccupée extends CoupIllégal {
    private static final long serialVersionUID = 1L;

    public CaseOccupée(int x, int y) {
        super(x, y);
    }
}
```

## Classifier des exceptions

Les cas d'erreur sont distingués

```
public class Dame {
    ...
    private void poser(int x, int y, Couleur c)
        throws CoupIllégal {
        if (0 < x || x >= TAILLE || 0 < y || y >= TAILLE)
            throw new CoordonnéeIllégale(x, y);

        if (!estLibre(x, y))
            throw new CaseOccupée(x, y);
        ...
    }
}
```

## Classifier des exceptions

Chaque type d'erreur peut être traité spécifiquement

```
public class Appli {
    public static void main(String[] args) {
        Dame dame = new Dame();

        while (!dame.finie()) {
            try {
                dame.jouer();
            }
            catch (CoordonnéeIllégale e) {
                System.out.println("à côté!!!");
            }
            catch (CaseOccupée e) {
                System.out.println("la case en (" +
                    e.getX() + ", " + e.getY() +
                    ") est déjà occupée");
            }
            catch (CoupIllégal e) {
                // traitement de tous les autres coups illégaux
            }
        }
    }
}
```

## Attrape-tout !!!

Une façon facile d'attraper toutes les exceptions

```
public class Appli {
    public static void main(String[] args) {
        try {
            // collez ici votre programme
        }
        catch (Exception e) {
            // Non, mon programme ne bug pas, il meurt en silence
        }
    }
}
```

## Précondition d'une méthode - Les règles préconisées par Sun

Les premières lignes d'une méthode sont généralement dédiées à la vérification de la validité de l'appel (valeur des paramètres, états de l'objet, etc.)

Le but est de détecter au plus tôt les erreurs et c'est particulièrement important pour les constructeurs

- ▶ Méthodes publiques
  - ▶ Un non respect du contrat doit être signalé à l'utilisateur par la levée d'une exception sous-classant `RuntimeException` (telle que `IllegalArgumentException`, `NullPointerException`, ou `IllegalStateException`)
  - ▶ Ainsi, l'utilisateur sera toujours prévenu d'une erreur (bloc `try/catch` ou interruption du programme)
- ▶ Méthodes privées
  - ▶ Le respect du contrat peut être testé par assertion
  - ▶ C'est au programmeur de s'assurer que tous les appels respecteront le contrat

## Éléments de syntaxe

```
try {  
    ...  
}  
catch (NullPointerException | ArithmeticException e) {  
    // même traitement pour des exceptions différentes  
}
```

```
try {  
    ...  
    return;  
}  
catch (NullPointerException e) {  
    ...  
}  
catch (ArithmeticException e) {  
    ...  
    return;  
}  
finally {  
    // bloc d'instructions tjs exécuté après le try/catch  
}
```

## Finally

Finally...