# Rappel des concepts fondamentaux du paradigme orienté objet

UNIVERSITÉ DE LORRAINE

Analyse et Conception de Logiciels

*Horatiu CIRSTEA*

# Exemples cours

Inspirés de

- Object-Oriented Software Engineering: Practical Software Development using UML and Java - Timothy C. Lethbridge, Robert Laganière
- Effective Java Programming Language Guide - Joshua Bloch

# Bonnes pratiques pour la programmation Java

# Classes et Interfaces

- Minimiser l'accessibilité
- Favoriser les objets immuables
- Préférer la composition à l'héritage
- Concevoir et documenter pour l'héritage
- Préférer les interfaces aux classes abstraites
- Définir des interfaces
- Hiérarchies de classes vs classes taggées

# Minimiser l'accessibilité

- pour une bonne conception **cacher** les *données internes* et les *détails d'implémentation*
- chaque **classe/méthode/attribut** aussi inaccessible que possible
- encapsulation ⇒ découplage ⇒ indépendance dans développement, tests, optimisations,...
- L'encapsulation n'entraîne pas de bonnes performances mais permet un réglage efficace des performances : les composants inefficaces peuvent être optimisés sans affecter les autres

# Minimiser l'accessibilité

- préférer les **attributs privés**

  ✓ exception possible: *constantes* (immuables)

  ✦ attribut pas final ou référence vers un immuable

    - on ne peut plus limiter le nombre de valeurs
    - on ne peut pas bloquer sa modification

  ✦ `protected` ⇒ **engagement wrt l'implémentation**

- préférer les **classes package-private** à public

# Minimiser l'accessibilité

- Une classe qui a des attributs **attributs public** ne permet pas
  - ✦ de changer la représentation interne
  - ✦ d'imposer des invariants
  - ✦ de réaliser des actions auxiliaire lors de l'accès
  - ➡ utiliser des getters (et setters si immuables)
- Moins grave si les attributs sont *immuables*
  - ✦ on peut garantir des invariants

# Favoriser les objets immuables

- toutes les informations contenues dans chaque instance sont fixes pendant toute la durée de vie de l'objet

- plus facile à ***concevoir***, à ***mettre en œuvre*** et à ***utiliser*** que les classes mutables; moins sujet aux erreurs
- une classe doit être immuable s'il n'y a pas une bonne raison pour qu'elle mutable

# Favoriser les objets immuables

- toutes les informations contenues dans chaque instance sont fixes pendant toute la durée de vie de l'objet
  - ➡ *exemples*: `String, Boolean, Integer,BigInteger,…`
- plus facile à ***concevoir***, à ***mettre en œuvre*** et à ***utiliser*** que les classes mutables; moins sujet aux erreurs
- une classe doit être immuable s'il n'y a pas une bonne raison pour qu'elle mutable

# Favoriser les objets immuables

- tous les attribut `private`
- tous les attributs `final`
- pas des méthodes qui modifie l'objet
- la classe ne peut pas être étendue
- accès exclusif aux composants mutables

➥ "petits" objets doivent être immuables

# Favoriser les objets immuables

```
public final class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x
        this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }

    public Point translation(int dx, int dy) {
        return new Point(x + dx, y + dy);
    }
    … // equals, hashCode, toString
}
```

# Favoriser les objets immuables

- ***simples***: toujours dans le même état (invariants garantis par le constructeur de la classe)
- ***thread-safe***: peuvent être partagés
- ***peuvent partager des internals*** (ex: `BigInteger`)
- des ***très bon composants*** pour d'autres objets (et éléments ensembles, clés tables, …)
- ***performance***: utiliser une classe *companion mutable* (ex: `StringBuilder`)

# Favoriser les objets immuables

```
public final class Complex {
    private final double re;
    private final double im;
    public Complex(double re,
                   double im) {
        this.re = re;
        this.im = im;
    }
    public double realPart() {
       return re;
    }
    public double imaginaryPart() {
       return im;
    }
    // functional approach
    public Complex plus(Complex c) {
       return new Complex(re + c.re,
                          im + c.im);
    }
    …
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Favoriser les objets immuables

```
public final class Complex {
    private final double re;
    private final double im;
    public Complex(double re,
                   double im) {
        this.re = re;
        this.im = im;
    }
    public double realPart() {
       return re;
    }
    public double imaginaryPart() {
       return im;
    }
    // functional approach
    public Complex plus(Complex c) {
       return new Complex(re + c.re,
                          im + c.im);
    }
    …
}
```

```
public class Complex {
    private final double re;
    private final double im;
    private Complex(double re,
                    double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex
      valueOf(double re, double im) {
        return new Complex(re, im);
    }
    // as before
    public double realPart() {
       return re;
    }
    …

}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# L'héritage et ses couplages

- l'héritage casse l'encapsulation: la sous-classe ***dépend des détails l'implémentation*** de la super-classe pour réaliser ses fonctions
- **des changements a posteriori de la super-classe peuvent casser le fonctionnement de la sous-classe**
  - ➡ la sous-classe doit évoluer en même temps que la super-classe (sauf si super-classe prévue pour héritage)

# L'héritage et ses couplages

- l'héritage casse l'encapsulation: la sous-classe ***dépend des détails l'implémentation*** de la super-classe pour réaliser ses fonctions
- **des changements a posteriori de la super-classe peuvent casser le fonctionnement de la sous-classe**
  - ➡ la sous-classe doit évoluer en même temps que la super-classe (sauf si super-classe prévue pour héritage)

➥ *composition* et forwarding

# L'héritage et ses couplages

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet() {}
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

```
public InstrumentedHashSet<String> s =
            new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# L'héritage et ses couplages

```
public class InstrumentedSet<E> extends ForwardingSet<E>
{
  private int addCount = 0;
  public InstrumentedHashSet(Set<E> s) {
       super(s);
   }
   @Override
   public boolean add(E e) {
       addCount++;
       return super.add(e);
   }
   @Override
   public boolean addAll(Collection<? extends E> c) {
      addCount += c.size();
      return super.addAll(c);
   }
   public int getAddCount() {
      return addCount;
   }
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# L'héritage et ses couplages

```
public class InstrumentedSet<E> extends ForwardingSet<E>
{
  private int addCount = 0;
  public InstrumentedHashSet(Set<E> s) {
       super(s);
   }
   @Override
   public boolean add(E e) {
       addCount++;
       return super.add(e);
   }
   @Override
   public boolean addA
       addCount += c.si
       return super.add
   }
   public int getAddCo
       return addCount
   }
}
```

```
public public class ForwardingSet<E>
                implements Set<E> {
   private final Set<E> s;
   public ForwardingSet(Set<E> s) { this.s = s; }
   public void clear() { s.clear(); }
   public boolean isEmpty() { return s.isEmpty(); }
   public boolean add(E e){ return s.addAll(c); }
   public boolean addAll(Collection<? extends E> {
      return s.addAll(c);
   }
   …
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# L'héritage et ses couplages

```
public class InstrumentedSet<E> extends ForwardingSet<E>
{
  private int addCount = 0;
  public InstrumentedHashSet(Set<E> s) {
       super(s);
   }
   @Override
   public boolean add(E e) {
       addCount++;
       return super.add(e);
   }
   @Override
   public boolean addA
       addCount += c.si
       return super.add
   }
   public int getAddCo
       return addCount
   }
}
```

```
public public class ForwardingSet<E>
                implements Set<E> {
   private final Set<E> s;
   public ForwardingSet(Set<E> s) { this.s = s; }
   public void clear() { s.clear(); }
   public boolean isEmpty() { return s.isEmpty(); }
   public boolean add(E e){ return s.addAll(c); }
   public boolean addAll(Collection<? extends E> {
      return s.addAll(c);
   }
   …
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Concevoir et documenter pour l'héritage

- documenter les méthodes ***overridable*** (not final/ private) et ***self-used***
  - ➡ on peut utiliser `@implSpec` en `JavaDoc`

**Class AbstractCollection<E>**

**isEmpty**

`public boolean isEmpty()`

Returns `true` if this collection contains no elements.

...

**Implementation Requirements:**

This implementation returns `size() == 0`.

# Concevoir et documenter pour l'héritage

- accepter certaines méthodes en `protected`

**Class AbstractCollection<E>**

**removeRange**

```
protected void removeRange (int fromIndex, int toIndex)
```

...

This method is called by the `clear` operation on this list and its subLists. Overriding this method to take advantage of the internals of the list implementation can *substantially* improve the performance of the `clear` operation on this list and its subLists.

**Implementation Requirements:**

This implementation gets a list iterator positioned before `fromIndex`, and repeatedly calls `ListIterator.next` followed by `ListIterator.remove` until the entire range has been removed. **Note: if `ListIterator.remove` requires linear time, this implementation requires quadratic time.**

# Concevoir et documenter pour l'héritage

- les constructeurs **ne doivent pas utiliser** des **méthodes** overridable ⇒ **utiliser méthodes *helper*** privées appelées dans constructeur (et dans méthodes overridable)

- **Quand permettre l'héritage ?**
  - ✓ classes abstraites (*skeletal implementations*)
  - – classes immuables
- **Comment bloquer ?**
  - ✓ final
  - ✓ constructeurs privés

# Préférer les interfaces aux classes abstraites

- classe existante facilement adaptable à une interface
- les interfaces idéales pour définir des “mixins” (des fonctionnalités peuvent être mixées avec l’existant)
- construction des frameworks non-hiérarchiques
- proposer des classes (abstraites) dérivées

➡ classe squelette (`Abstract`*`List`*`, Abstract`*`Set`*`,...`)

➡ classe de base (squelette pas abstract)

# Définir des interfaces

- **méthodes `default` pour une extension plus facile**
  - ✦ les classes qui implémentaient l'interface continuent de fonctionner mais peuvent générer des erreur au runtime
  - ✦ peuvent casser des invariants des implémentations
  - ✦ éviter de les ajouter a posteriori
- **les interfaces doivent définir des *types***
  - ✦ il ne faut pas les utiliser juste pour déclarer des constantes

# Hiérarchies de classes vs classes taggées

```
class Figure {
  enum Shape { RECTANGLE, CIRCLE };
  final Shape shape;
  // RECTANGLE
  double length;
  double width;
  // CIRCLE
  double radius;

  Figure(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
  }
  …
```

```
  double area() {
    switch(shape) {
      case RECTANGLE:
        return length * width;
      case CIRCLE:
        return PI*(radius*radius);
      default:
        throw new
    AssertionError(shape);
    }
  }
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Hiérarchies de classes vs classes taggées

```
class Figure {
  enum Shape { RECTANGLE, CIRCLE };
  final Shape shape;
  // RECTANGLE
  double length;
  double width;
  // CIRCLE
  double radius;
```

```
abstract class Figure{
  double area();
}
```

…

```
double area() {
  switch(shape) {
    case RECTANGLE:
      return length * width;
    case CIRCLE:
      return PI*(radius*radius);
    default:
      throw new
```

```
class Circle extends Figure {
  final double radius;
  @Override
  double area() { … }
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Hiérarchies de classes vs classes taggées

➡ **verbose, error-prone, inefficaces**

- code boilerplate (enum, attribut tag, switch, …)
- plusieurs implémentations en une classe
- empreinte mémoire des objets plus importante
- le constructeur doit initialiser les attributs en fonction du tag
- attributs ne peuvent pas être final si pas initialisés dans le constructeur

# Hiérarchies de classes vs classes taggées

```
class Figure {
  enum Shape { RECTANGLE, CIRCLE };
  final Shape shape;
  // RECTANGLE
  double length;
  double width;
  // CIRCLE
  double radius;

  Figure(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
  }
  …
```

```
  double area() {
    switch(shape) {
      case RECTANGLE:
        return length * width;
      case CIRCLE:
        return PI*(radius*radius);
      default:
        throw new
    AssertionError(shape);
    }
  }
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Hiérarchies de classes vs classes taggées

```
class Figure {
  enum Shape { RECTANGLE, CIRCLE };
  // Tag field - the shape of this figure
  final Shape shape;
  // These fields are used only if shape is CIRCLE
  final double radius;
  // These fields are used only if shape is RECTANGLE
  final double length;
  final double width;
  // Constructor for circle
  Figure(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
  }
  …
```

```
abstract class Figure{
  double area();
}
```

```
class Circle extends Figure {
  final double radius;

  @Override
  double area() {
      …
  }
}
```

```
    }
  }
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Création d'objets

- Utilisation de static factory methods
- Singletons
- Eviter la création d'objets inutiles

# Static factory methods

- **ont des noms**
  - ✓ code plus facile à lire
  - ✓ plusieurs méthodes avec le même profil (et noms différents) contrairement à un unique constructeur avec un profil donné

```
/** Constructs a randomly generated positive BigInteger that is
    probably prime, with the specified bitLength. **/
BigInteger(int len, int cert, Random rnd)

/** Constructs a randomly generated, … */
static BigInteger probablePrime(int len, Random rnd)
```

# Static factory methods

- ne doivent pas obligatoirement construire un nouvel objet et évite ainsi de créer inutilement des objets

```
Boolean.valueOf(boolean)
```

- **peuvent garantir:**
  - ✓ que la classe est un singleton
  - ✓ que la classe ne peut pas être instanciée
  - ✓ qu'il n'existe pas deux instances (immuables) identiques (et donc une comparaison efficace)

# Static factory methods

- **peuvent retourner un objet d'un sous-type de la classe**
  - des interfaces comme types de retour
  - eg. **`java.util.Collections`** (classe sans instances pour l'interface **`Collection`**)

```
/* Returns a synchronized (thread-safe) list … */
static <T> List<T> synchronizedList(List<T> list)
…
/* Returns an unmodifiable view of the set */
static <T> Set<T> unmodifiableSet(Set<? extends T> s)
```

# Static factory methods

- le type de l'objet retourné peut varier d'un appel à un autre (ou d'une version à une autre)
  - ➡ `java.util.EnumSet`
- le type de l'objet retourné ne doit pas exister au **moment où la classe est développée**
  - ➡ les bases du framework service-provider (service interface, provider registration API, service access API), eg. JDBC

# Static factory methods

- la classe ne peut pas être héritée si pas de constructeur `public` ou `protected`
- peuvent être plus difficiles à trouver ⇒ utiliser des noms plus ou moins standard

✓ `Date d = Date.from(instant);`

✓ `Set<Color> colours = EnumSet.of(RED, BLUE);`

✓ `BigInteger prime = BigInteger.valueOf(5);`

✓ `getInstance, newInstance, getType, newType, type`

# Singletons

- constructeur `private`

```
public class JamesBond {
  public static final JamesBond
           INSTANCE = new JamesBond();

  private int power;

  private JamesBond() {…}

  public void attack() {…}
}
```

```
public class JamesBond {
  private static final JamesBond
           INSTANCE = new JamesBond();
  public static JamesBond
    getInstance() { return INSTANCE;}

  private int power;

  private JamesBond() {…}

  public void attack() {…}
}
```

# Singletons

- constructeur `private`

```
public class JamesBond {
  public static final JamesBond
           INSTANCE = new JamesBond();

  private int power;

  private JamesBond() {…}

  public void attack() {…}
}
```

```
public enum JamesBond {
  INSTANCE;

  private int power;

  private JamesBond() {…}

  public void attack() {…}
}
```

```
public class JamesBond {
  private static final JamesBond
           INSTANCE = new JamesBond();
  public static JamesBond
    getInstance() { return INSTANCE;}

  private int power;

  private JamesBond() {…}

  public void attack() {…}
}
```

# Dependency injection

```
public class SpellChecker {
    private final Lexicon dictionary = ...;
    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);
    public boolean isValid(String word) { ... }
   ...
}
```

Singleton

*Effective Java 3rd ed., Joshua Bloch,  Pearson Education Inc.*

# Dependency injection

```
public class SpellChecker {
   private final Lexicon dictionary = ...;
   private SpellChecker(...) {}
   public static INSTANCE = new SpellChecker(...);
   public boolean isValid(String word) { ... }
  ...
}
```

Singleton

```
public class SpellChecker {
   private static final Lexicon dictionary = ...;
   private SpellChecker(...) {}
   public boolean isValid(String word) { ... }
  ...
}
```

Utility class

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Dependency injection

```
public class SpellChecker {
   private final Lexicon dictionary = ...;
   private SpellChecker(...) {}
   public static INSTANCE = new SpellChecker(...);
   public boolean isValid(String word) { ... }
  ...
}
```

Singleton

```
public class SpellChecker {
   private static final Lexicon dictionary = ...;
   private SpellChecker(...) {}
   public boolean isValid(String word) { ... }
  ...
}
```

Utility class

```
public class SpellChecker {
   private final Lexicon dictionary;

   public SpellChecker(Lexicon dictionary) {
      this.dictionary = Objects.requireNonNull(dictionary);
   }
   public boolean isValid(String word) { ... }
  ...
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Eviter les objets inutiles

- Eviter plusieurs instances d'un objet immuable

```
String s = new String("big");
```
😱

```
String s = “big";
```
😍

- Utiliser factory methods

```
new Boolean(“true”);
```
😱

```
Boolean.valueOf(“true”);
```
😍

- Créer une seule fois des objets “couteux”

```
public static boolean isRomanNumeral(String s) {
       return s.matches(""^(?=.)M*(C[MD]|D?C{0,3})"
           + “(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Eviter

```
public static boolean isRomanNumeral(String s) {
        return s.matches("“^(?=.)M*(C[MD]|D?C{0,3})"
            + “(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

- Eviter plusieurs instances d’un objet immuable

```
String s = new String("big");
```
😱

```
String s = “big";
```
😍

- Utiliser factory methods

```
new Boolean(“true”);
```
😱

```
Boolean.valueOf(“true”);
```
😍

- Créer une seule fois des objets “couteux”

```
public class RomanNumerals {
  private static final Pattern ROMAN =
    Pattern.compile(“^(?=.)M*(C[MD]|D?C{0,3})"
       + “(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

  static boolean isRomanNumeral(String s) {
    return ROMAN.matcher(s).matches();
} }
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Obsolete references

```
public class Stack {

  private Object[] elements;
  private int size = 0;
  private static final int C = 16;

  public Stack() {
    elements = new Object[C];
  }

  public Object pop() {
    if (size == 0)
      throw new StackEx();
    return elements[--size];
  }
```

```
  public void push(Object e) {
    ensureCapacity();
    elements[size++] = e;
  }

  private void ensureCapacity(){
    if (elements.length == size)
      elements = Arrays.copyOf(
                   elements,
                   2 * size + 1);
  } }
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Méthodes communes

- Respecter le contrat de `equals`
- Réécrire `hashCode` avec `equals`
- `Implémenter Comparable` et `compareTo`

# Le contrat de `equals`

- **Propriétés**
  - ✓ réflexivité
  - ✓ symétrie
  - ✓ transitivité
  - ✓ consistence (même résultat avec mêmes params)
  - ✓ **`x.equals(null)==false`**

# Le contrat de equals

- symétrie

```
public final class CaseInsensitiveString {
        private final String s;
        …
        @Override public boolean equals(Object o) {
            if (o instanceof CaseInsensitiveString)
                return s.equalsIgnoreCase(
                    ((CaseInsensitiveString) o).s);
            if (o instanceof String)
                return s.equalsIgnoreCase((String) o);
            return false;
        }
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Le contrat de equals

- transitivité

```
public class Point {
    private final int x, y;
    public boolean equals(Object o) {
      if (!(o instanceof Point))
        return false;
      Point p = (Point)o;
      return p.x == x && p.y == y;
    }
}

public public class ColorPoint extends Point {
    private final Color color;
    public boolean equals(Object o) {
       if (!(o instanceof ColorPoint))
          return false;
       return super.equals(o) &&
              ((ColorPoint) o).color == color;
    }
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Le contrat de equals

- transitivité

```
public class Point {
    private final int x, y;
    public boolean equals(Object o) {
      if (!(o instanceof Point))
        return false;
      Point p = (Point)o;
      return p.x == x && p.y == y;
    }
}

public public class ColorPoint extends Point {
    private final Color color;
    public boolean equals(Object o) {
       if (!(o instanceof Point))  // pour la symétrie
          return false;
       if (!(o instanceof ColorPoint)) // Point
           return o.equals(this);
       return super.equals(o) &&
              ((ColorPoint) o).color == color;
    }
}
```

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Le contrat de `equals`

- aucun objet n'est égal à `null`

```
public boolean equals(Object o) {
  if (o == null)
    return false;  // pas nécessaire
  if (!(o instanceof Point))
    return false;
  Point p = (Point)o;
  return p.x == x && p.y == y;
}
```

- surtout pour éviter des `NullPointerException`

*Effective Java 3rd ed., Joshua Bloch, Pearson Education Inc.*

# Le contrat de `equals`

- **Etapes dans la conception de `equals`**
  - utiliser == pour vérifier si même instance
  - utiliser **`instanceof`** pour vérifier si bon type
  - utiliser **`cast`** pour l'argument
  - comparer les attributs significatifs

😱 `public boolean equals(MyClass o)`

# Le contrat de `equals`

- Ne pas redéfinir **`equals`** si pas nécessaire
  - chaque instance est unique
  - pas besoin d'une égalité logique (e.g. **`Random`**)
  - la super-classe le fait bien
  - si on peut savoir que equals sera jamais utilisé (dans une classe package-private)

# Redéfinir `hashCode`

- le contrat de `hashCode`
  - retourne la même valeur pendant une exécution si les infos utilisées dans **`equals`** changent pas
  - les objets **`equals`** doivent avoir le même hashcode
  - *résultats différents pour des objets pas* **`equals`** *améliorent les performances des collections*

# Redéfinir `hashCode`

- si `equals` est redéfini, redéfinir `hashCode`

```
public final class Person {
   private final String name;
   @Override
   public boolean equals(Object obj) {…}
…
}
```

```
Map<Person,String> m = new HashMap<>();
m.put(new Person(“James”), “smart”);
String s = m.get(new Person(“James”));
```

# Redéfinir `hashCode`

- ne pas utiliser des méthodes trop simples

  ```
  public int hashCode() { return 5; }
  ```

- une bonne fonction de hachage a tendance à produire des codes de hachage inégaux pour des instances inégales
- utiliser tous les attributs utilisés pour equals (pas forcement les dérivés)
- combiner avec une fonction

  result = 31 * result + c

# Comparable et compareTo

- retourne <0, 0, >0 si this <,=,>
- `ClassCastException` si incomparable
- `sgn(x.compareTo(y))==-sgn(y.compareTo(x))`
- `(x.compareTo(y)>0 && y.compareTo(z)>0) ⇒ x.compareTo(z)>0`
- `x.compareTo(y)==0 ⇒ sgn(x.compareTo(z))==sgn(y.compareTo(z))`

# Méthodes

- Vérifier la validité des paramètres
- Effectuer des copies défensives si besoin
- Utiliser le overloading avec attention
- Retourner des arrays ou collections vides, pas des nulls
- ...

# Questions ?