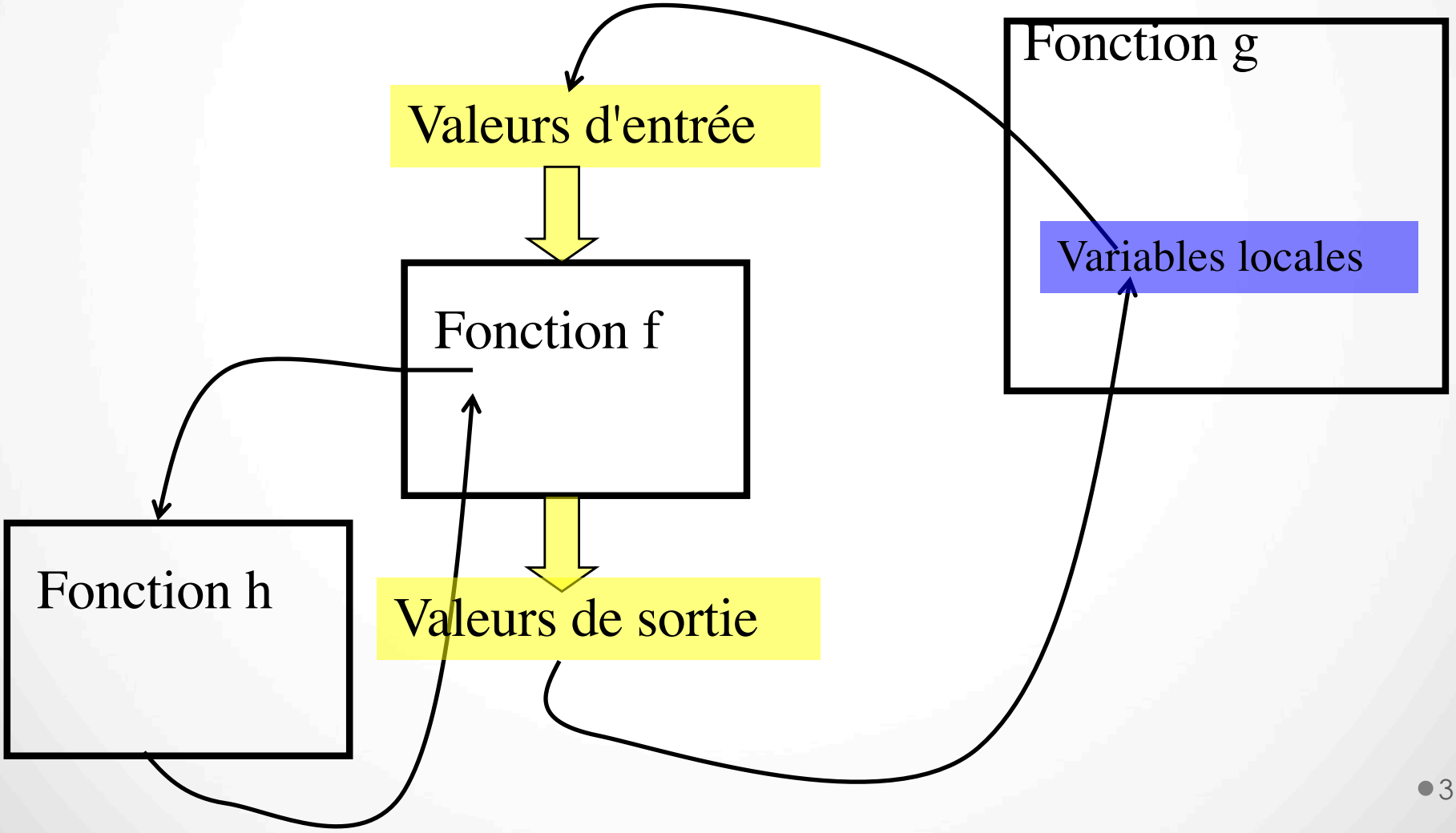
# Le Modèle Objet

# Introduction

- Constats :
  - croissance de la complexité des projets
  - besoin de gain de productivité
- Qualités attendues :
  - modularité / résistance aux modifications
  - réutilisabilité
  - lisibilité et compréhensibilité
  - abstraction des données
- Solution : **L'OBJET**

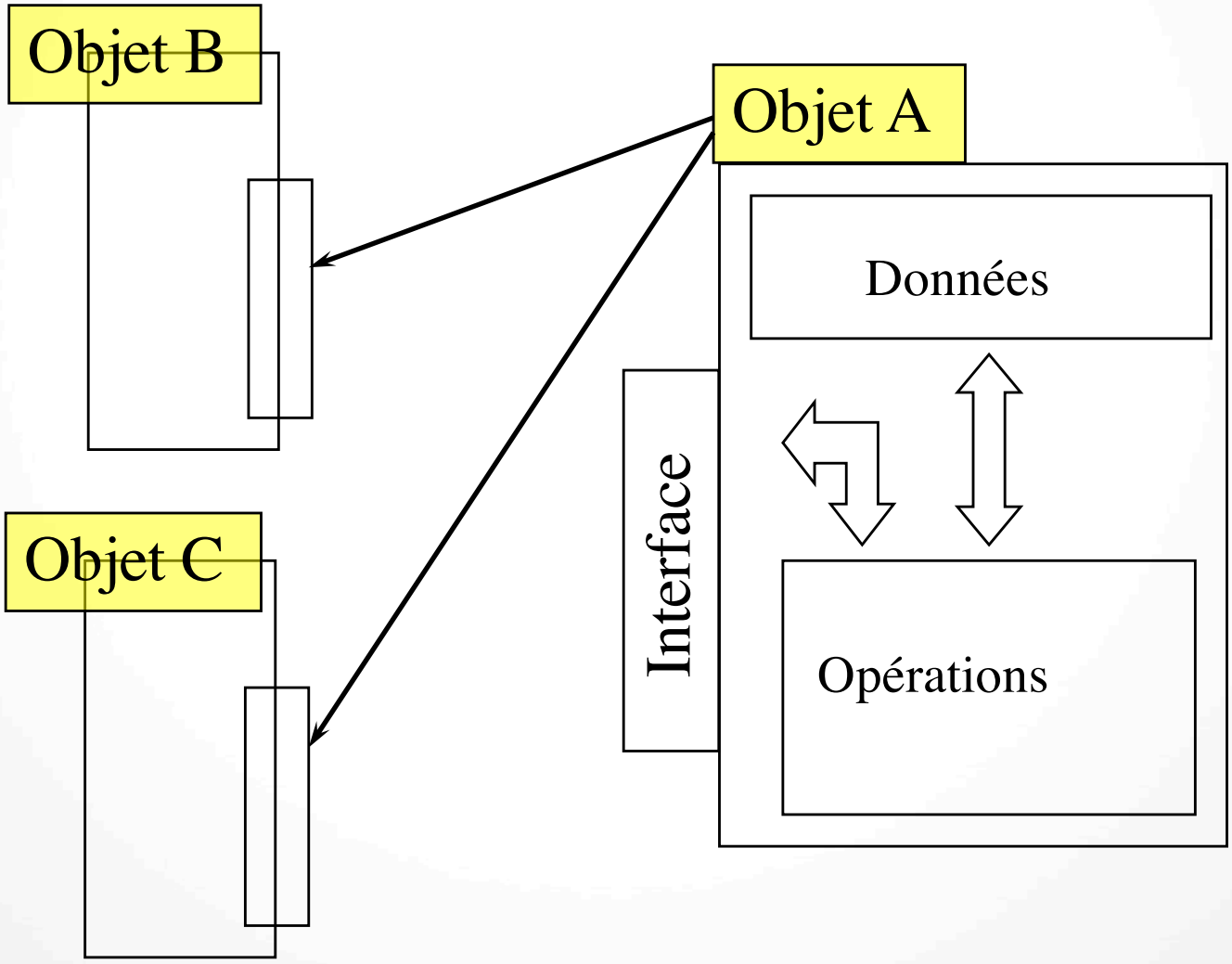
# Avant l'Objet : le procédural

Séparation entre les données et les traitements qui les manipulent



# Orienté Objet

Encapsulation des données et des traitements associés.
Communication par messages au travers de l'interface des objets.



# Modèle Objet

- Un objet est constitué de données.
  - Ex. un rectangle possède une longueur, une largeur, …
  - Les données d'un objet sont stockées dans ses attributs.

- Un objet manipule ses données pour effectuer des opérations :
  - Ex. un rectangle peut se dessiner, se translater, se redimensionner, …
  - Les opérations d'un objet sont réalisées par l'exécution des méthodes correspondantes.
  - L'exécution d'une méthode est déclenchée par la réception d'un message envoyé par un autre objet. (Il y a donc un objet par lequel tout commence…)

- La structure des attributs et des méthodes d'un objet sont décrits dans sa classe.

# Les classes et instances

- Une classe décrit un modèle de données (les attributs) et de comportement (les méthodes)
  - Les attributs et méthodes décrits dans une classe sont appelés les membres de la classe
  - Une classe peut être vue comme un « patron » décrivant (et permettant de créer) de nombreux objets qui fonctionnent conformément au même modèle.
    - Chat : décrit les entités ayant 4 pattes, 1 queue, et sachant miauler
    - Voiture : décrit les entités ayant des portes, des roues et sachant rouler

- Tout objet est l'instance d'une classe
  - Instancié (créé) à partir du modèle décrit dans sa classe, il possède les même attributs (liste, noms et types) et les même opérations (signatures) que les autres instances issues de la même classe
  - Instances de Chat : tom, garfield, kitty, leChatDuVoisin, …

# Point de vue d'informaticien

- Les classes correspondent au programme
  - description des structures de données (liste, nom et type des attributs)
  - description des opérations (liste, nom et algorithme des méthodes)
- L'informaticien écrit des programmes, donc des classes
  - un programme est écrit une seule fois, mais peut être exécuté en plusieurs exemplaires (et à volonté)
  - chaque exécution est une instance qui fonctionne conformément à son modèle (le programme)
- Les objets correspondent à l'exécution du programme
  - chaque instance s'exécute conformément à sa classe (son modèle/programme)
  - on peut créer ne nombreuses instances à partir de la même classe
- L'exécution d'un programme orienté objet correspond à un ensemble d'objets qui interagissent conformément au programme décrit dans leur classe

# Structure des classes

- Une classe est un agrégat d'attributs et de méthodes

```
visibilité class NomClasse {

        visibilité Type nomAttribut;
        …

        visibilité Type nomMethode(paramètres) {
                …
        }

        …

}
```

# Exemple de classe

```java
public class Circle {

    private double x, y, rayon ;

    /** Calcul de l'aire
     * @return l'aire du cercle
     */
    public double getArea(){
        return Math.PI * Math.pow(rayon, 2) ;
    }

    /** Translation sur l'axe x
     * @param value : decalage sur l'axe
     */
    public void translateX(double value){
        x = x + value ; // x += value ;
    }

}
```

# Instanciation avec new

- Pour créer un **objet**, on **instancie** sa classe en appliquant l'opérateur **new**.

- Grâce à **new**, une nouvelle **instance** de cette classe est allouée en mémoire. On peut alors l'utiliser.

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

    Circle c = new Circle() ; // instanciation d'un objet de type Circle

    c.translateX(50); // execution d'une méthode (envoie de message à l'OBJET)

    double aire = c.getArea() ; // appel de méthode avec récupération du résultat

}
```

# Les packages

- Un package regroupe un ensemble de classes sous un même espace de nommage.
- Le mot clé **package** indique à quel package appartient la ou les classe(s) de l'unité de compilation (le fichier).
- **package** doit être la première instruction du fichier.
- Les noms des packages suivent le schéma : **name.subname** (ex. **java.util**)
- Une classe **Watch** appartenant au package **time.clock** doit se trouver dans le fichier **time/clock/Watch.class** (**time** et **clock** étant des répertoires)

# Les packages

- En dehors du package (dans un autre), les noms des classes sont : **packageName.className**

  Ex. **java.util.Vector v = new java.util.Vector()**;

- L'instruction **import packageName** permet d'utiliser des classes sans les préfixer par leur nom de package
  - Ex. **import java.util.*** (toutes les classes du package **java.util** peuvent être utilisées sans mentionner le nom du package comme indiqué au point précédent)

- Les APIs sont organisées en package (**java.lang**, **java.io**, ...)

- Les répertoires contenant les packages doivent être présents dans la variable d'environnement **CLASSPATH**

# Les packages

- Seules les classes publiques d'un package sont utilisable à partir d'un autre package

**Circle** : classe publique du package **graph**

```
package graph;

public class Circle { // classe publique !

    private double x, y, rayon ;

    /** Calcul de l'aire
     * @return l'aire du cercle
     */
    public double getArea(){
        return Math.PI * Math.pow(rayon, 2) ;
    }

    /** Translation sur l'axe x
     * @param value : decalage sur l'axe
     */
    public void translateX(double value){
        x = x + value ; // x += value ;
    }

}
```

Autre package : utilisation du vrai nom de **Circle**…

```
package tests;

public class Test1 {
    public static void main(String[] args) {

        // instanciation d'une classe d'un autre package
        graph.Circle c = new graph.Circle() ;

    }
}
```

… ou importation pour alléger l'écriture

```
package tests;

import graph.Circle ;

public class Test1 {
    public static void main(String[] args) {

        // instanciation d'une classe d'un autre package
        Circle c = new Circle() ;

    }
}
```

# Accès aux membres

- Rappel : les classes définissent des « boites noires »
  - encapsulation des attributs et des méthodes
  - accès gérés au niveau de l'interface
- L'accessibilité des membres est défini au niveau de la classe :
  - Privé (private) : inaccessible hors de l'objet
  - Protégé (protected) : inaccessible hors de la hiérarchie de classes (cf. héritage)
  - Friendly () : inaccessible en dehors du package
  - Publique (public) : totalement accessible
- Principe d'encapsulation :
  - On essaiera toujours de limiter l'accès au maximum
  - private > protected > *friendly* > public

# Accès aux membres

```java
package graph;

public class PointBizarre {

    int x ; // friendly (NB: devrait être privé)
    private int y ; // privé

    // Methode publique
    public void setXandY(int abscisse, int ordonnee){
        x = abscisse ;
        y = ordonnee ;
    }

}
```

```java
package graph ; // Même package que PointBizarre

public class RectangleBizarre {

    private PointBizarre coords = new PointBizarre();

    public int longueur, largeur ; // Devraient être privés !

    public void setCoords(int abscisse, int ordonnee){
        coords.x = abscisse ; // OK : x est friendly dans PointBizarre
        coords.y = ordonnee ; // PAS OK : y est privée dan PointBizarre
    }

}
```

```java
package tests ; // Package différent de graph

import graph.PointBizarre;
import graph.RectangleBizarre;

public class Test1 {
    public static void main(String[] args) {

        PointBizarre p = new PointBizarre() ;
        p.x = 10 ; // PAS OK : x friendly mais autre package
        p.y = 20 ; // PAS OK : y privé
        p.setXandY(10, 20); // OK : setXandY est publique !

        RectangleBizarre rectangle = new RectangleBizarre() ;
        rectangle.longueur = 50 ; // OK : longueur est public

    }
}
```

# L'encapsulation

- L'encapsulation présente divers avantages :
  - Encapsuler les données permet de :
    - les protégées des accès intempestifs, garantissant l'intégrité de l'objet
    - déclencher des actions spécifiques lorsqu'on y accède
  - Les clients d'une abstraction ne dépendent pas de la réalisation de l'abstraction, mais seulement de sa spécification
    - les signatures des méthodes constituent l'interface d'utilisation d'un objet
    - l'implémentation des méthodes peut évoluer sans que cela impacte les programmes qui les utilisent (qui y font appel)

# Accès contrôlés : getters / setters

Utiliser l'**encapsulation** pour contrôler les accès !

```
package graph;

import javafx.scene.paint.Color;

public class Point {

    private int x, y ;
    private Color color = Color.BLACK ;
    private boolean black = true ;

    public int getX() { return x; }
    public void setX(int x) { this.x = x; }

    public int getY() { return y; }
    public void setY(int y) { this.y = y; }

    public Color getColor() { return color; }
    private void setColor(Color color) { this.color = color; }

    public boolean isBlack() {
        return black;
    }

    void setBlack(boolean black) {
        this.black = black;
        if(this.black) setColor(Color.BLACK);
        else setColor(Color.WHITE);
    }
}
```

Le mot clé **this**
est utilisé par un objet
pour se référencer lui même

# L'identité

- Deux objets (instances) peuvent avoir exactement les mêmes valeurs dans leurs attributs (le même état) et proposer les mêmes méthodes, ils ne se confondent pas l'un avec l'autre.
  - Ex. Deux voitures à la sortie de l'usine
- Réciproquement, les valeurs contenues dans les attributs d'un objet peuvent changer (changement d'état), l'objet lui-même ne change pas d'identité.
  - Ex. Une voiture repeinte
- Le langage objet fournit un moyen de désigner un objet en tant qu'élément unique, indépendamment de son état : référence, pointeur, identificateur...
  - Une variable de type objet contient une référence ou null

# null

- Une variable objet ne peut être utilisée si son contenu est **null**
  - Dans ce cas il n'y a pas d'objet pointé, donc pas d'exécution du code associé possible

**Classe Droite avec des attributs de type objet (Point) vides**

```
package graph;

public class Droite {

    private Point p1, p2 ; // non initialisés donc contiennent null

    public void translationX(int decalage){
        p1.setX( p1.getX() + decalage );
        p2.setX( p2.getX() + decalage );
    }

}
```

**Programme principal**

```
public static void main(String[] args) {
    Droite d = new Droite() ;
    d.translationX( decalage: 10);
}
```

**Erreur à l'exécution**

```
Exception in thread "main" java.lang.NullPointerException
    at graph.Droite.translationX(Droite.java:8)
    at tests.Test2.main(Test2.java:14)

Process finished with exit code 1
```

# Initialisation d'objet : constructeur

- Un constructeur est une méthode qui a pour objectif d'initialiser l'objet en cours de création
  - Il porte le nom de la classe et retourne (implicitement) une instance de la classe
  - L'appel au constructeur est déclenché par l'instruction new
- Toute classe possède par défaut un constructeur sans paramètre
  - C'est celui qui a été (implicitement) utilisé dans nos exemples précédents
  - Le constructeur par défaut peut être redéfini
- Une classe peut avoir plusieurs constructeurs qui diffèrent par le nombre et la nature de leurs paramètres
  - Le choix du constructeur est fait derrière l'instruction new en donnant les paramètres à utiliser

# Constructeur(s)

```java
public class Droite {

    private Point p1, p2 ; // non initialisés donc contiennent null

    public Droite(int x1, int y1, int x2, int y2){
        p1 = new Point() ; // instanciation de l'objet référencé par p1
        p1.setX(x1) ; p1.setY(y1);

        p2 = new Point() ; // instanciation de l'objet référencé par p2
        p2.setX(x2); p2.setY(y2);

        System.out.println("Appel au constructeur avec paramètres !");
    }

    /** Surcharge du constructeur par défaut (sans paramètres)
     * en faisant appel au constructeur avec paramètres pour éviter de dupliquer du code
     */
    public Droite(){
        this(0,0,0,0) ; // si on utilise this (appel à un autre constructeur), ça doit être la 1ere instruction
        System.out.println("Appel au constructeur sans paramètres !");
    }

    public void translationX(int decalage){...}
}
```

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

    Droite d1 = new Droite() ; // Utilisation du (nouveau) constructeur par défaut
    d1.translationX(50); // Ne pose plus de problème car les points sont bien instanciés

    Droite d2 = new Droite(10, 8, 25, 12) ; // Utilisation du constructeur paramétré

}
```

# Passage des paramètres

- Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :
  - par copie pour les types primitifs
  - par référence pour les objets

```
public class Test {

    private static void modifier(int i, StringBuffer chaine){
        i++ ; // type primitif
        chaine.append("d") ; // type objet
    }

    public static void main(String[] args) {

        int a = 0 ;
        StringBuffer message = new StringBuffer("abc") ;

        modifier(a, message);

        System.out.println(a); // affiche (toujours) 0
        System.out.println(message); // affiche "abcd" !

    }

}
```

# Constructeur par recopie

```
public class Carre {

    private Point point;

    private int longueur ;

    public Carre(int x, int y, int longueur){
        point = new Point() ;
        point.setX(x); point.setY(y);
        this.longueur = longueur ;
    }

    public Carre(Carre carre){

        this.longueur = carre.longueur ;// OK: primitif

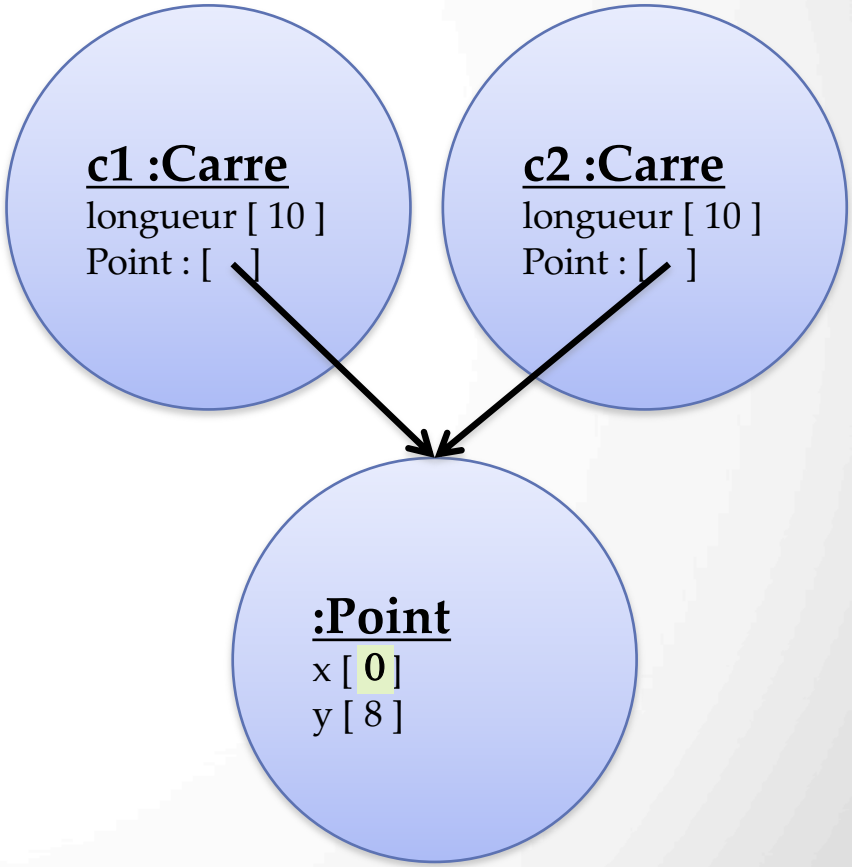
        this.point = carre.point ;// ! objet = reference !

    }

    public void setX(int x){
        this.point.setX(x);
    }

}
```

```
public static void main(String[] args) {
    Carre c1 = new Carre(5, 8, 10) ;
    Carre c2 = new Carre(c1) ;
    c2.setX(0);
}
```



# Constructeur par recopie

```java
public class Carre {

    private Point point;

    private int longueur ;

    public Carre(int x, int y, int longueur){
        point = new Point() ;
        point.setX(x); point.setY(y);
        this.longueur = longueur ;
    }

    public Carre(Carre carre){

        this.longueur = carre.longueur ;// OK: primitif

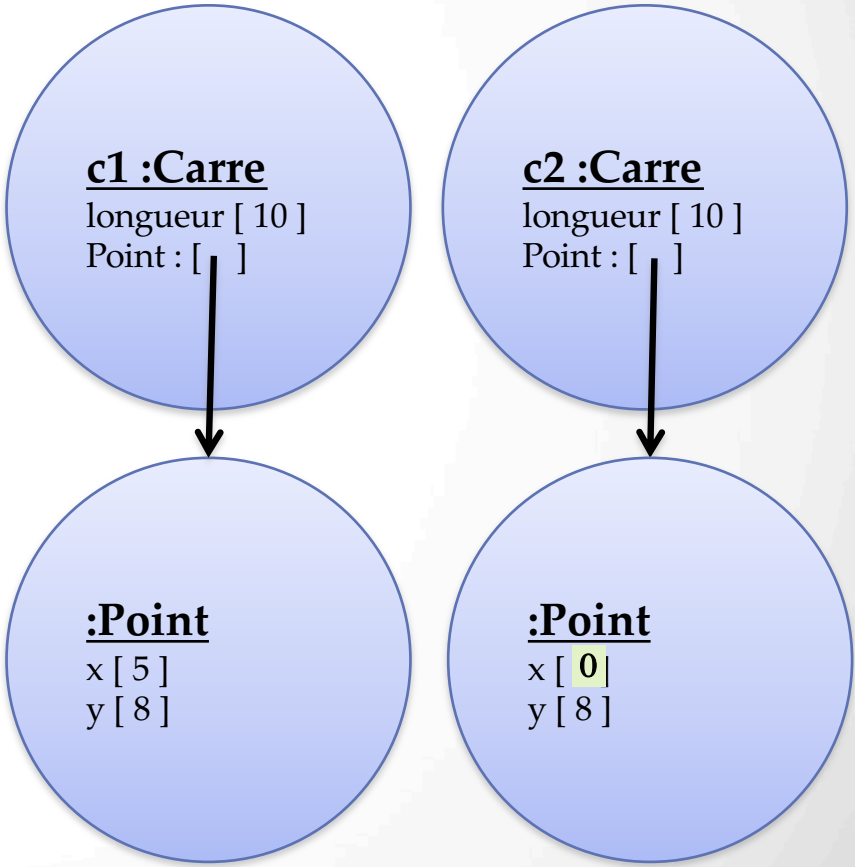
        // Céation d'une nouvelle instance
        this.point = new Point() ;

        // recopie des attributs
        this.point.setX(carre.point.getX());
        this.point.setY(carre.point.getY());
    }

    public void setX(int x){
        this.point.setX(x);
    }

}
```

```java
public static void main(String[] args) {
    Carre c1 = new Carre(5, 8, 10) ;
    Carre c2 = new Carre(c1) ;
    c2.setX(0);
}
```



# static

- En Java, **les classes sont aussi des objets**... !!!
  - Les objets « classe » sont instanciés lors du chargement de la classe dans la JVM (en général au lancement du programme)
  - La JVM ne peut pas contenir plusieurs objets représentant la même classe
  - **Chaque objet « classe » est donc unique dans l'environnement**

- Etant des objets, les classes ont leur propres membres
  - Les membres d'une classe sont définis en tant que **static**
  - L'accès aux membres **static** se fait directement sur l'objet « classe »
    - Pas d'instanciation puisque déjà instancié

- **Le code d'une classe contient donc 2 niveaux** de définition
  - Le niveau d'instance qui concernera les instances de la classe
  - Le niveau de classe (static) qui concerne la classe elle-même

# static : analogie

- Les classes sont des modèles
  - Ex. un document papier contenant un patron pour créer des **Clio**
  - Le modèle n'est pas l'instance : le papier n'est pas une Clio
  - Le modèle a des attributs (static) :
    - **static couleur** du papier
      - La couleur du papier n'a rien à voir avec la couleur des voitures
  - Le modèle a des méthodes (static) :
    - **static plier()** le papier
      - Plier le papier n'est pas plier une voiture…
- Les instances sont créées selon le modèle
  - Ex. des voitures correspondant aux instruction du modèle **Clio**
  - La classe **Clio** décrit les attributs d'**instance**
    - **couleur** de la voiture
      - Ce n'est pas la couleur du papier, et chaque voiture (instance) a la sienne
    - **accelerer()** : s'adresse à une voiture particulière (une instance)

# static

```java
public class Etudiant {

    private static int nombre_etudiants_crees = 0 ;

    private String nom ;
    private int numCarte;

    private static void nouvelleInscription(){
        nombre_etudiants_crees++ ;
    }

    public static int getNbEtudiants(){
        return nombre_etudiants_crees ;
    }

    public Etudiant(String nom){
        nouvelleInscription();
        this.nom = nom ;
        this.numCarte = getNbEtudiants() ;
    }

    public String getNom() {
        return nom;
    }

    public int getNumCarte() {
        return numCarte;
    }
}
```

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

    Etudiant sacha, helmut, omer ;

    sacha = new Etudiant("Touille") ;
    helmut = new Etudiant("Hardelpik") ;
    omer = new Etudiant("Dalors") ;

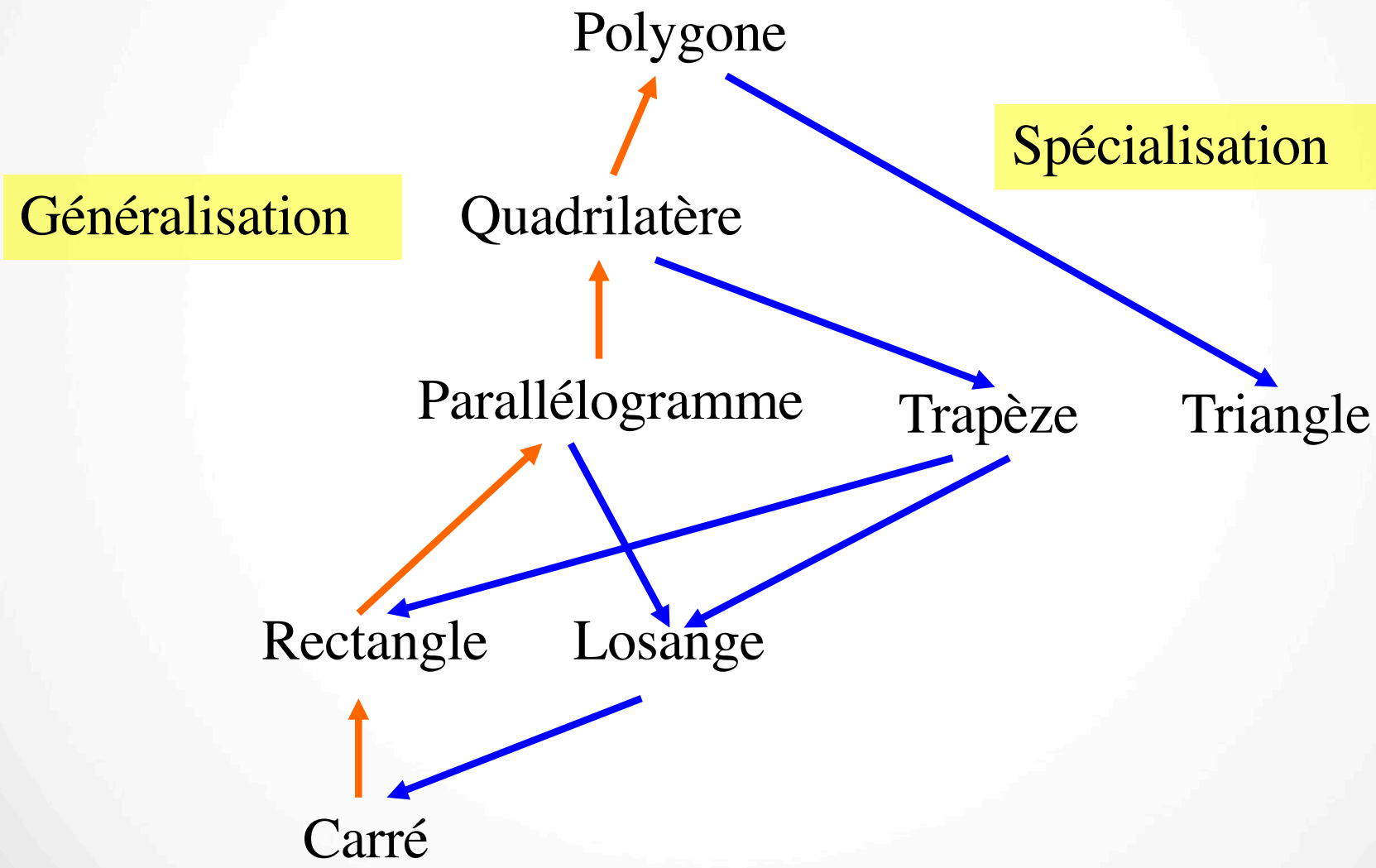
    int NB = Etudiant.getNbEtudiants() ; // appel direct sur la classe
    System.out.println(NB); // affiche 3

    System.out.println(helmut.getNom()); // affiche "Hardelpik"
    System.out.println(helmut.getNumCarte()); // affiche "2"

    System.out.println(sacha.getNom()); // affiche "Touille"
    System.out.println(sacha.getNumCarte()); //affiche 1

}
```

# L'héritage - Exemple



# L’héritage

- L’héritage est la technique la plus utilisée pour réaliser la généralisation ou la spécialisation
- Construire une classe à partir d’une ou plusieurs autres classes, en partageant des attributs, des méthodes et parfois des contraintes, au sein d'une hiérarchie de classes
- Chaque sous-classe incorpore, ou hérite, tous les membres de sa **super-classe** et y ajoute ses propres et propriétés

# Héritage

- Une classe ne peut hériter (**extends**) que d'une seule classe.
  - Pas d'héritage multiple
  - **class nom_classe extends nom_classe_mère {… }**
- Le mot clé **super** désigne la classe mère
- Les classes **final** ne peuvent pas être redéfinies dans les sous-classes.

# Héritage

```java
public class Personne {

    private String nom ;

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

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

}
```

```java
public final class Etudiant extends Personne {

    private int numCarte;

    public Etudiant(String nom, int numCarte){
        super(nom) ;
        this.numCarte = numCarte ;
    }

    public int getNumCarte() {
        return numCarte;
    }
}
```

```java
public class ProfInfo extends Prof {

    private static final String MATIERE = "Info" ;

    public ProfInfo(String nom){
        super(nom, MATIERE) ;
    }
}
```

```java
public class Prof extends Personne {

    private String matiere ;

    public Prof(String nom, String matiere){
        super(nom) ;
        this.matiere = matiere ;
    }

    public String getMatiere() {
        return matiere;
    }
}
```

```java
Etudiant helmut = new Etudiant("Hardelpik", 123456) ;
ProfInfo greg = new ProfInfo("Bourguin") ;

System.out.println(helmut.getNom()); // OK: Personne
System.out.println(helmut.getNumCarte()); // OK : Etudiant

System.out.println(greg.getNom()); // OK : Personne
System.out.println(greg.getMatiere()); // OK : Prof
```

# Héritage : surcharge

- Lorsqu'une méthode est réécrite dans une sous-classe, on parle de **surcharge**

- Par exemple :
  - si on définit un **Polygone** avec un tableau de points, pour le dessiner il suffit de faire un tracé entre chaque point
  - si on définit ensuite la classe **Carre** comme une sous-classe de polygone, on peut redéfinir la méthode **dessiner** car l'algorithme peut être plus efficace
  - => on surcharge la méthode **dessiner** en la spécialisant avec les propriétés spécifique du carré

- Une sous-classe peut accéder aux membres de sa super-classe seulement s'ils sont au minimum protected

# Héritage : surcharge

```java
public class Rectangle {

    protected int longueur, largeur ;

    public Rectangle(int longueur, int largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    public long getArea(){
        System.out.println("getArea() de Rectangle");
        return longueur * largeur ;
    }

}
```

```java
public class Carre extends Rectangle{

    public Carre(int longueur) {
        super(longueur, longueur);
    }

    @Override
    public long getArea() {
        return (long)Math.pow(longueur, 2) ;
    }
}
```

```java
Rectangle rectangle = new Rectangle(10, 20) ;
Carre carre = new Carre(5) ;

long aire ;
aire = rectangle.getArea() ; // getArea() de Rectangle
aire = carre.getArea() ; // getArea() de Carre

rectangle = carre ; // OK : un carre est un rectangle

rectangle.getArea() ; // getArea() de Carre !!!
```

# Héritage : java.lang.Object

- Toutes les classes héritent implicitement de java.lang.Object
  - Une classe sans héritage a en réalité un ... extends Object
  - Toute classe hérite donc de fait des membres définis dans Object

All Methods | Instance Methods | Concrete Methods

| Modifier and Type | Method and Description |
|---|---|
| protected Object | clone()<br>Creates and returns a copy of this object. |
| boolean | equals(Object obj)<br>Indicates whether some other object is "equal to" this one. |
| protected void | finalize()<br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class<?> | getClass()<br>Returns the runtime class of this Object. |
| int | hashCode()<br>Returns a hash code value for the object. |
| void | notify()<br>Wakes up a single thread that is waiting on this object's monitor. |
| void | notifyAll() |
| String | toString()<br>Returns a string representation of the object. |

**Appelée automatiquement lors d'un print**

| | |
|---|---|
| | Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. |
| void | wait(long timeout, int nanos)<br>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed. |

# Surcharge de toString()

```java
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Locale;

public class Personne {

    private String nom, prenom ;
    Calendar naissance ;
    int numSecu ;

    public Personne(String nom, String prenom, int numSecu) {
        this.nom = nom;
        this.prenom = prenom;
        this.numSecu = numSecu ;
    }

    public void setNaissance(Calendar naissance) {
        this.naissance = naissance;
    }

    @Override
    public String toString() {
        String chaine = String.format(Locale.FRANCE, "%s %s [%3$tA %3$te %3$tB %3$tY]", prenom, nom.toUpperCase(), naissance) ;
        return chaine ;
    }

    // On peut mettre le main de test dans la classe elle-même...
    public static void main(String[] args) {
        Personne greg = new Personne("Bourguin", "Grégory", 007) ;

        Calendar calendar = GregorianCalendar.getInstance(Locale.FRANCE) ;
        calendar.set(1971, Calendar.OCTOBER, 9);
        greg.setNaissance(calendar);

        System.out.println(greg); // "Grégory BOURGUIN [samedi 9 octobre 1971]"
    }
}
```

# Final

- Le mot clé **final** signifie « changement interdit »

- Il interdit :
  - la surcharge d'une méthode ( **final void m(…) {…}** )
  - la spécialisation d'une classe ( **final class A {…}** )
  - la modification d'un attribut ou d'un argument d'une méthode ( **final int I** )

# Final

```
class A {
    .....
}
```

```
class B {

    final int i = 0; // Initialized final
    final int j ;
    final A a ;

    B() {
            j = 1; // Initialize
            a = new A();
    }

    B(int x) {
            j = x; // Initialize
            a = new A() ;
    }
}
```

**Si les attributs *final* ne sont pas initialisés lors de leur déclaration, ils doivent alors être initialisés dans le constructeur.**

**Ils ne pourront plus être modifiés par la suite.**