

Télécharger le corrigé C10 sur les SWINGs

<https://bit.ly/2TMHooH>

CHAPITRE 02

PROGRAMMATION ORIENTÉE OBJET EN

JAVA

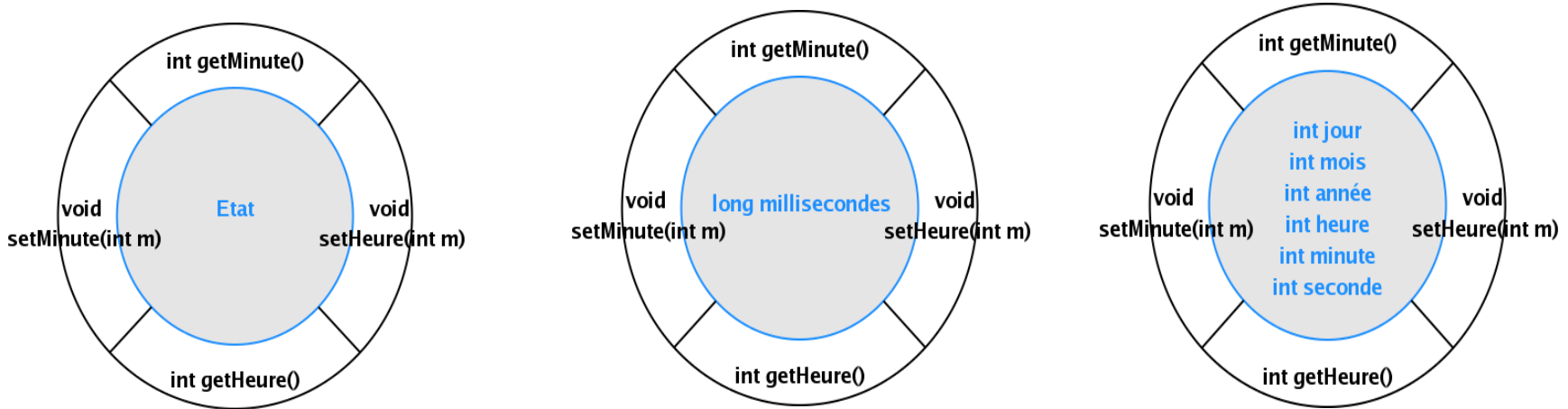
<https://bit.ly/2TEIKRM>

2.1 CLASSES ET OBJETS

Principe d'encapsulation

- Un objet = des données (attributs) + algorithmes (méthodes)
- Principe d'encapsulation : l'accès aux données (privées) ne peut se faire que par l'intermédiaire de l'**interface** (méthodes publiques).
- Les modifications des structures de données (ou attributs) n'affectent pas les programmes qui utilisent l'objet.

➔ **Réutilisation.**



Classes & Objets

- Une classe décrit, définit, implémente un type d'objet :
 - ses **attributs** et ses **méthodes**
 - son **interface** (*public*) et sa partie *private*.
- *Elle doit être utilisable, réutilisable, évolutive :*
 - *principe d'encapsulation*
 - *interface bien pensée*
- Une classe est un modèle de définition pour des objets ayant
 - même structure (même ensemble d'attributs),
 - même comportement (mêmes opérations, méthodes),
 - une sémantique commune.
- Une classe n'est pas un objet.
- **Un objet est l'instanciation d'une classe** (et une seule), une représentation dynamique, « vivante ».

Type de l'objet = sa classe

```
public class Etudiant {  
    String nom;  
    int promo;  
    String traiterExam(String enonce)  
    { .. }  
}  
  
public class Prof {  
    String nom;  
    String faireSujetExam()  
    { .. }  
    int corrigerExam(String copie)  
    { .. }  
}
```

Attributs

Méthodes

The diagram illustrates the relationship between class members and their types. It features two Java class definitions: `Etudiant` and `Prof`. The `Etudiant` class has attributes `String nom` and `int promo`, and a method `String traiterExam(String enonce) { .. }`. The `Prof` class has attributes `String nom` and `String faireSujetExam()`, and methods `int corrigerExam(String copie) { .. }`. A green arrow labeled 'Attributs' points to the attribute declarations in both classes. A red arrow labeled 'Méthodes' points to the method declarations in both classes.

Structure d'une Classe

- Attributs
 - La structure de données définissant et caractérisant l'état interne de chaque objet de la classe.
 - Le contenu, l'information nécessaire
- Méthodes
 - Définition du comportement interne/externe de l'objet, ce qu'il peut faire.
 - Les méthodes *public* définissent l'interface.
- Les constructeurs
 - Des méthodes particulières,
 - Ils construisent l'instance d'une classe (construction et initialisation) et,
 - Renvoient une référence sur l'objet construit.

Contenu d'une classe

- *Les variables d'instance définissent l'état d'une instance.*

```
public class Etudiant {  
    String nom;  
    int promo;  
}
```

- *Méthodes de construction*
 - *Constructeur : Exécuté à la création d'une nouvelle instance tout de suite après le chargement en mémoire.*
- *Méthodes d'altération*
 - *initialisent, modifient ou annulent la valeur des variables d'instance.*
- *Méthodes d'accès (Getters et Setters)*
 - *lisent ou renseignent les valeurs des variables d'instance.*

Exemple de classe

```
public class Etudiant {  
    // Les attributs sont privés  
    private String nom;  
    private int promo;  
    // Les constructeurs  
    Etudiant(String n, int p){  
        nom = n;  
        promo = p;  
    }  
    // Méthodes  
    public String  
        traiterExam(String  
            enonce){...}  
}
```

```
public class AdminFac {  
    public static void main(String[]  
        args){  
        // Déclaration et  
        // Instantiation d'un objet  
        Etudiant et = new Etudiant(  
            "Robert",5);  
    }  
}
```

Getters et Setters

```
public class Etudiant {  
    // les attributs sont privés  
    private String nom;  
    private int promo;  
    // Les constructeurs  
    Etudiant(){  
        nom = « Toto »; promo = 0;  
    }  
  
    Etudiant(String n, int p){  
        nom = n; promo = p;  
    }  
  
    // Assesseurs  
    public String getNom(){ return nom; }  
    public int getPromo(){ return promo; }  
    public void setPromo(int p){ promo = p;}  
    // et les algo nécessaires.  
    public String traiterExam( String enonce){...}  
}
```

Constructeur par défaut

Déclaration (classe):

```
public class MyClass {  
    /* attributs */  
    private long myVar;  
    /* méthodes */  
    public void incr() {  
        myVar++;  
    }  
}
```

Création (objet) :

```
MyClass myInstance = new MyClass();
```

myInstance est l'adresse de l'objet créé.

Les **objets** sont toujours implicitement passés par adresse à des fonctions.

MyClass() : Constructeur par défaut (si le programmeur ne définit pas de constructeur)...

➤ garantit l'initialisation (à des valeurs par défaut : 0, false, null, etc.) des variables de la classe.

Constructeur défini

Déclaration (classe):

```
public class MyClass {  
    /* attributs */  
    private long myVar;  
    /* constructeurs */  
    public MyClass(long var){  
        myVar = var;  
    }  
    /* méthodes */  
    public void incr() {  
        myVar++;  
    }  
}
```

Création (objet) :

```
MyClass myInstance = new MyClass(99);
```

```
MyClass myInstance = new MyClass();
```

Le constructeur par défaut MyClass() n'est pas fourni si le programmeur définit au moins un constructeur.

this

Surcharge :

```
public class MyClass {  
    /* attributs */  
    private long myVar;  
    /* constructeurs */  
    public MyClass(long var){  
        myVar = var;  
    }  
    public MyClass(){  
        this(0);  
    }  
    /* méthodes */  
    public void incr() {  
        this.myVar++;  
    }  
}
```

Création (objet) :

```
MyClass myInstance = new MyClass(99);
```

Ou

```
MyClass myInstance = new MyClass();
```

this(...) appelle un constructeur (en fonction des arguments).
Ne peut être appelé qu'au début d'un constructeur.

this.myVar : référence

Déclaration vs Création

- En Java, les objets sont manipulés exclusivement avec des **références**. Une référence est un pointeur.

- Création d'une référence sur un objet de type Integer :

Integer s;

- La création d'une référence, n'implique pas la création d'un objet. Les objets doivent être créés par :

s = new Integer();

- Ainsi est créé un objet de type Integer. On y accède via la référence s.
- Plusieurs références peuvent pointer sur le même objet :

Integer s2 = s;

- Il y a toujours qu'un seul objet, mais on a deux références.

Modificateurs d'accès

Dans un objet l'accès et l'utilisation d'un attribut ou d'une méthode est contrôlé :

- **public** : tout le monde peut y accéder, entre autre un utilisateur le « client programmeur ».
 - Définit l'interface de la classe.
- **private** : accès interne à l'objet
 - Selon le principe d'encapsulation, les attributs doivent être « private »
- le « **friendly** package access » : public pour les classes du même package
- **protected** : public pour les classes qui héritent

Méthodes statiques : déclaration

Pour une méthode statique :

static <typeRetour> nomMethode(<liste de paramètres>)
{ <corps de la méthode> }

```
public class Exemple {
```

```
    public static void printFloat(String s, float f) {  
        System.out.println(s + " = " + f);  
    }
```

```
}
```

```
public class ExempleApp {
```

```
    public static void main(String args[]) {  
        Exemple.printFloat("Résultat=", 0.5);  
    }
```

```
}
```


Méthodes

- Les Méthodes sont l'équivalent des fonctions. Elles permettent de :
 - *factoriser du code,*
 - *structurer le code,*
 - *servir de « sous programmes utilitaires » aux autres méthodes de la classe.*
- En java, plusieurs types de méthodes :
 - *opérations sur les objets (envoi d'un message),*
 - *opérations statiques (méthodes de classe)*
exemples `Math.random()` ;
`LectureClavier.lireEntier()` ;

Méthodes d'objet

- Par opposition aux méthodes de classe (**static**)
- La signature s'écrit de la même manière au sein de la classe, sans le mot clé **static** :
 <typeRetour> nomDeMethode(liste des params formels)
- Nécessite l'existence d'un objet pour être invoquée :
 monObjet.laMethode(liste des params effectifs);
- Le constructeur est une méthode particulière : pas de valeur de retour, mais des paramètres.
- Toute classe proprement faite possède son/ses constructeur(s) (au moins 1) qui garantissent la création et l'initialisation propre des objets.

```
public static void main(String[] args) {  
    Integer a = new Integer(3);  
    int x = 3;  
    // x+=a; est incorrect  
    x+=a.intValue();  
    System.out.println("x = " + x);  
}
```

x = 6;

2.2 SPÉCIFICITÉS

La méthode equals()

- Égalité entre objet et non entre référence

```
Point p1 = new Point(2,1);
Point p2 = new Point(2,1);
if (p1==p2){...} // Ici c'est faux
if (p1.equals(p2)){...} // OK si equals est bien redéfini
p1 = p2 // Co-référence
if (p1==p2){...} // Vrai désormais
```

- Très utile pour les String
 - If (myString == "Hello") {...} Toujours Faux !!!
 - If (myString.equals("Hello")) {...} OK
- Si les 2 objets o1 et o2 ont des références r1 et r2 comme champs, il faut que r1==r2 pour que o1.equals(o2) et non pas que r1.equals(r2)
- Réflexive, symétrique et transitive
- null.equals(null) mais aucune autre instance oi telle que oi.equals(null)
- Pour les classes que vous créez, vous êtes responsable.

Classes enveloppes - 1/3

- Les classes enveloppes fournissent une contrepartie sous forme de classes aux types primitifs.
- Les classes enveloppes sont final, donc non surchargeables

Nombres entiers	int	Integer
Nombres entiers longs	long	Long
Nombres décimaux	float	Float
Nombres décimaux en double précision	double	Double
Booléens	boolean	Boolean
Octets	byte	Byte
Caractères (Unicode)	char	Character
Type vide	void	Void

Classes enveloppes – 2/3

- Quelques méthodes de la classe Integer par exemple :

```
public int intValue();
```

- Renvoie la valeur entière de l'objet (type int).

```
public static int parseInt(String s);
```

- Renvoie la valeur entière de l'objet si l'objet spécifié représente la chaîne d'un nombre entier en base 10.

```
public static Integer.valueOf(String  
s)
```

- Renvoie un objet Integer si la chaîne s représente un nombre entier en base 10.

- Exemple :

```
Integer unInteger = new Integer(100);  
int d = unInteger.intValue();
```

Classes enveloppes – 3/3

- Fonction equals() pour comparer
- Beaucoup de fonctions statiques (pas besoin de faire de new pour les utiliser)
- Float, Byte, Integer peuvent tous retourner une valeur int, float, etc. car héritage de Number
- Fonction compareTo()

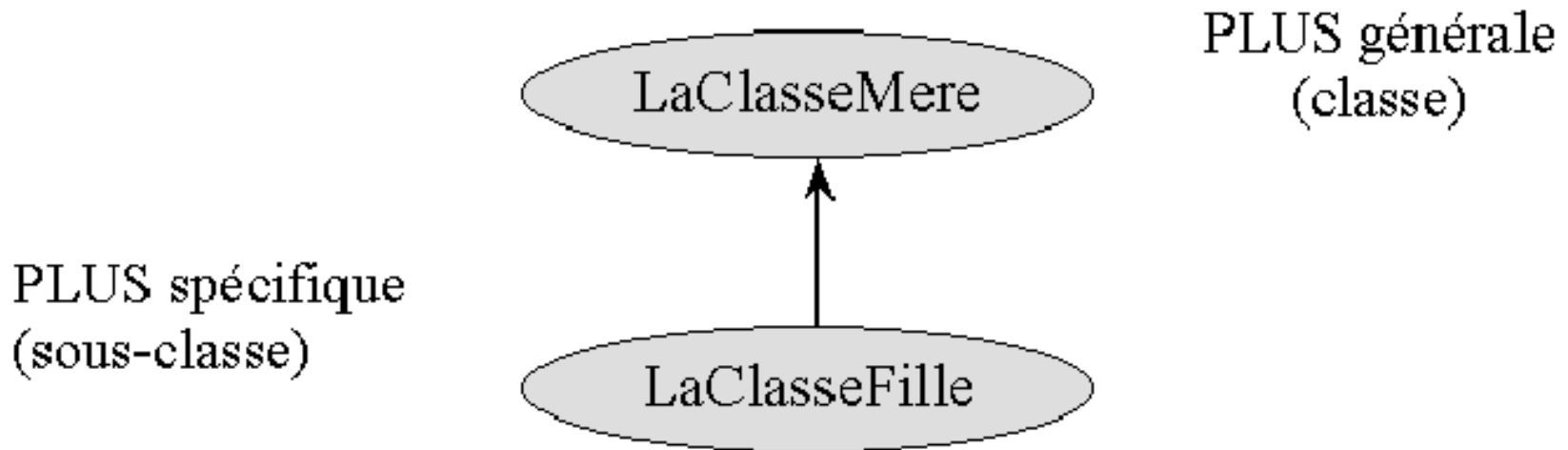
2.3 HÉRITAGE EN JAVA

Réutilisation ?

- Comment utiliser une classe comme matériau pour concevoir une autre classe répondant à de nouveaux besoins ?
- Quels sont les attentes de cette nouvelle classe ?
 - besoin des «services» d'une classe existante (les données structurées, les méthodes, les 2).
 - faire évoluer une classe existante (spécialiser, ajouter des fonctionnalités, ...)
- Quelle relation existe entre les 2 classes ?
- Dans une conception objet, des relations sont définis, des règles d'association et de relation existent.
- Un objet fait appel un autre objet.
- Un objet est crée à partir d'un autre : il hérite.

Héritage

```
public class LaClasseFille extends LaClasseMere{...}
```



```
class PointAvecUneCouleur extends Point
```

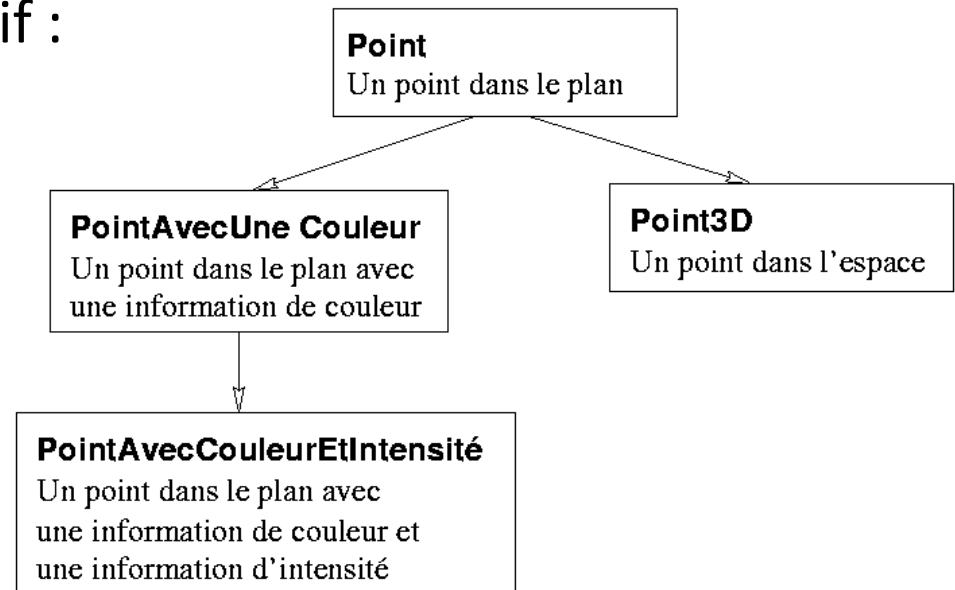
- En java, on n'hérite que d'une seule classe.
- Un objet de ClasseFille peut faire tout ce que sait faire un objet de ClasseMere
- Il le fait différemment et/ou il peut plus.

Exemple en Javadoc

The screenshot shows a web browser window displaying the Javadoc for the `JButton` class in the `javax.swing` package. The browser's address bar shows the file path `file:///Applications/Utilities/Java/J2SE%205.0/docs/api/index.html`. The page title is `JButton (Java 2 Platform SE 5.0)`. The left sidebar contains a navigation pane with a tree view of the Java 2 Platform Standard Ed. 5.0 API, including packages like `java.applet` and `javax.swing`, and a list of classes such as `JDesktopPane`, `JDialog`, `JEditorPane`, `JFileChooser`, `JFormattedTextField`, `JFormattedTextField.Abstr`, `JFrame`, `JInternalFrame`, `JInternalFrame.JDesktopic`, `JLabel`, `JLayeredPane`, `JList`, `JMenu`, `JMenuBar`, `JMenuItem`, `JMException`, `JMRuntimeException`, `JMXAuthenticator`, `JMXConnectionNotification`, and `JMXConnector`. The main content area has a navigation bar with links for `Overview`, `Package`, `Class` (selected), `Use`, `Tree`, `Deprecated`, `Index`, and `Help`. Below this, there are links for `PREV CLASS`, `NEXT CLASS`, `FRAMES`, and `NO FRAMES`. The `SUMMARY` section lists `NESTED`, `FIELD`, `CONSTR`, and `METHOD`. The `DETAIL` section lists `FIELD`, `CONSTR`, and `METHOD`. The `Class JButton` section shows the class hierarchy: `java.lang.Object` extends `java.awt.Component`, which extends `java.awt.Container`, which extends `javax.swing.JComponent`, which extends `javax.swing.AbstractButton`, which extends `javax.swing.JButton`. The `All Implemented Interfaces:` section lists `ImageObserver`, `ItemSelectable`, `MenuContainer`, `Serializable`, `Accessible`, and `SwingConstants`. The `Direct Known Subclasses:` section lists `BasicArrowButton` and `MetalComboBoxButton`. The `public class JButton` section shows the class declaration: `public class JButton` extends `AbstractButton` implements `Accessible`. The `An implementation of a "push" button. See How to Use Buttons, Check Boxes, and Radio Buttons in The Java Tutorial for information and examples of using buttons.`

Hiérarchie en Héritage

- L'héritage est une notion transitive
- Hiérarchie d'héritage : l'ensemble des classes dérivées d'un parent commun.
- Attention ce n'est pas bijectif :
 - un *Point3D* est un *Point*.
 - Le contraire est faux.
- *is-like* relation



Chaîne d'héritage : le chemin menant d'une classe vers ses ancêtres.

Concept d'héritage

- Concept fondamental des langages objet.
- Dériver une nouvelle classe à partir d'une classe existante en récupérant ses propriétés.
- Avantage
 - Réutilisation : enrichissement de champs et méthodes de classes existantes.
 - Spécialisation d'une classe existante sans la modifier.
 - Pas de limitation en profondeur.
- Contrainte
 - Héritage simple : une classe ne peut hériter que d'une seule classe.
 - Toutes les méthodes et champs ne sont plus regroupées dans le même fichier

Exemple d'héritage

```
import Point;
public class PointAvecUneCouleur extends Point
{
    int couleur; // x et y sont hérités
    public PointAvecUneCouleur(int x, int y, int couleur) // un constructeur
    {
        super(); // appel au constructeur de la classe mère
        setX(x);
        setY(y);
        setCouleur(couleur);
    }

    public PointAvecUneCouleur(int couleur) // un constructeur
    {
        super(); // appel au constructeur de la classe mère
        setCouleur(couleur);
    }

    public void setCouleur(int couleur)
    {
        this.couleur=couleur; // désigne l'objet courant
    }
}
```

Exemple d'utilisation des constructeurs

- `PointAvecUneCouleur p1,p2,p3;`

- `x=10, y=100, couleur=1000`

- `p1=new PointAvecUneCouleur(10,100,1000);`

- `x=y=0` (valeur par défaut), `couleur=5`

- `p2=new PointAvecUneCouleur(5);`

- Mais ce qui est impossible :

- `p3=new PointAvecUneCouleur();`

- `p4=new Point (10,100,1000);`

protected

- Dans la classe *Point* pour *setX* on a 3 possibilité :

```
public    void setX (double nouveauX){...}  
private  void setX (double nouveauX){...}  
protected void setX (double nouveauX){...}
```

- ***public*** : un objet *PointAvecUneCouleur* peut fixer l'abscisse, ... également tout le monde.
- ***private*** : plus personne ne peut, y compris un objet *PointAvecUneCouleur*.
- ***protected*** : la solution ici, les objets d'une classe héritière ont accès aux membres *protected*

Transtypage et héritage 1/2

- Le transtypage consiste à convertir un objet d'une classe en objet d'une autre classe
- Vers une «super-class» (*upcast*), c'est toujours possible : on peut toujours transtyper vers une super-classe, plus pauvre en informations.

```
Point3D unPoint3D = new Point3D( x, y, z );
```

```
Point unPoint = (Point) unPoint3D; // une projection sur xOy
```

- La mention du transtypage est facultative, mais recommandée pour la lisibilité.
- Attention : unPoint3D est de type *Point3D* mais aussi *Point*.
- « un Point3D est un Point mais pas le contraire. »
- Un message envoyable à un objet, l'est aussi pour tous les objets d'une classe héritière :

```
PointAvecUneCouleur p = new PointAvecUneCouleur();
```

```
p.translater(1,3); // Méthode définit au niveau de Point
```

Transtypage et héritage 2/2

- *Le comportement d'un objet «hérité» est augmenté par rapport à un objet d'une classe mère.*
- Vers sous-classes (*downcast*)
 - Impossible sauf si appartenance : pour transtyper vers une sous-classe, utiliser le mot-clé réservé **instanceof**.

```
if( unPoint instanceof Point3D) {  
    unPoint3D = (Point3D) unPoint;  
    unPoint3D.projection3D(...);  
}
```

- La mention du transtypage est obligatoire, dans ce cas.
- Quand le compilateur retourne une erreur " bad class ... ", ne pas transtyper

Surcharge et redéfinition (*overload* vs *override*)

```
public class BaseClasse {  
    ...  
    public void maMethode(int i){...}  
}
```

Surcharge

```
public class filleClasse1 {  
    ...  
    public void maMethode(double i){...}  
}
```

Pour un objet *filleClasse1* :

- 2 méthodes *maMethode*

Redéfinition

```
public class filleClasse2 {  
    ...  
    public void maMethode(int i){...}  
}
```

Pour un objet *filleClasse2* :

- une seule *maMethode*

Constructeurs et héritage

- L'héritage permet la réutilisation de code. Le constructeur est concerné.
- Possibilité d'invoquer une méthode héritée : la référence **super**.
- Invocation d'un constructeur de la super-classe :
 - `super(<params du constructeur>)`
- L'utilisation de `super` ressemble à celle de `this`
- **L'appel à un constructeur de la super classe doit toujours être la première instruction dans le corps du constructeur.**
 - Si ce n'est pas fait, il y a appel implicite : `super()`.
 - Lors de la création d'un objet, les constructeurs sont invoqués en remontant de classe en classe la hiérarchie, jusqu'à la classe `Object`.
 - Attention, l'exécution se fait alors en redescendant dans la hiérarchie.
- Un constructeur d'une classe est toujours appelé lorsqu'une instance de l'une de ses sous classes est créée.

Extends

```
public class DerivedClass extends ParentClass {  
  
}
```

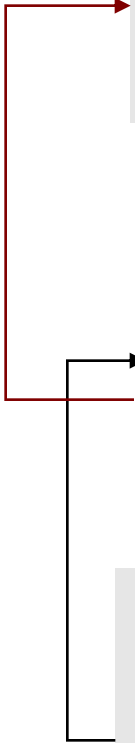
DerivedClass = classe dérivée

ParentClass = classe mère

- La classe dérivée hérite de tout ce qui est public et protected de ParentClass
- Peut redéfinir les méthodes public et protected
- Héritage simple uniquement : une classe ne peut hériter que d'une seule classe

Héritage et constructeurs - suite

```
public class Object
{
    public Object()
    {...}
}
```



```
public class Point extends Object
{
    public Point(double i, double j)
    {...}
}
```

```
public class PointAvecUneCouleur extends Point
{
    public PointAvecUneCouleur(int i, int j, int c){
        super(i,j);
        couleur=c;
    }
}
```

Conseils pour l'héritage

- Placer les informations communes dans la superclasse : principe d'économie et de non redondance de l'information
- N'utiliser l'héritage que lorsque c'est pertinent : c'est-à-dire lorsque toutes les informations de la surclasse sont bien utilisées par une sous-classe
- Utiliser le polymorphisme plutôt que des tests sur le type : éviter un code tel que :

```
if( x instanceof Classe1 ) {  
    x.methode1();  
}  
else if( x instanceof Classe2 ) {  
    x.methode2();  
}
```

this vs. super

```
public class ParentClass{

    protected int x;
    protected int y;

    public ParentClass(){this(0,0);}
    public ParentClass(int x, int y){
        this.x=x;
        this.y=y;
    }
    public void setX(int x){this.x = x;}
    public void setY(int y){this.y = y;}
    public int getX(){return x;}
    public int getY(){return y;}

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

```
public class DerivedClass extends ParentClass{

    protected int z;
    private int x;

    public DerivedClass(){this(0,0,0);}
    public DerivedClass(int x, int y, int z){
        super(x,y);
        this.z=z;
    }

    public void setZ(int z){this.z = z;}
    public int getZ(){return z;}
    public String toString() {
        return super.toString()+" z="+z;
    }
}
```


Exemple JFrame au tableau

final

```
public final class NonDerivableClass {  
  
}
```

Les classes déclarées **final** ne sont pas dérivables :

```
public class DerivedClass extends NonDerivableClass {  
  
}
```

INTERDIT



Ne pas confondre avec **final** pour les constantes : `final String NOM;`

final

```
public class MyClass {  
    ...  
    public final void method(){ ...}  
    ...  
}
```

MyClass est dérivable :

```
public class DerivedClass extends MyClass {  
  
}
```

OK

Les méthodes déclarées **final** ne sont pas redéfinissables dans les classes dérivées :

```
public class DerivedClass extends MyClass {  
    ...  
    public final void method(){ ...} →  
    ...  
}
```

INTERDIT

interface

- Similaire à une classe.
- Simple **description pour définir un** comportement **commun et abstrait** entre plusieurs classes (~ prototype).
- Toutes les méthodes sont abstraites par défaut. Toutes les variables sont **final static** par défaut.
- Elle ne s'instancie pas, il n'y a pas de code, pas de constructeur.
- Se déclare comme une classe, mais avec le mot clé « **interface** » (au lieu de « **class** »).
- Le mot clé « **implements** » permet de dire qu'une classe suit les règles de cette interface.

```
public interface NameOfInterface
{
    //final, static variables
    //abstract methods
}
```

interface

```
public interface Printable {  
    int MIN = 5; // par défaut, une constante statique  
    void print(); // par défaut, une méthode abstraite  
}
```

```
class MyClass implements Printable {  
    public void print(){System.out.println("Hello");}  
  
    public static void main(String args[]){  
        MyClass obj = new MyClass();  
        obj.print();  
    }  
}
```

interface

```
public interface Printable {  
    int MIN = 5; // par défaut, une constante statique  
    void print(); // par défaut, une méthode abstraite  
}
```

```
public interface Showable {  
    void show();  
}
```

```
class MyClass implements Printable, Showable {  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("display")}  
  
    public static void main(String args[]){  
        MyClass obj = new MyClass();  
        obj.print();  
        obj.show();  
    }  
}
```

interface

```
public interface AnotherInterface extends Printable,  
Showable{  
  
    void anotherMethod();  
  
}
```

```
class MyClass implements AnotherInterface{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("display")}  
    public void anotherMethod(){System.out.println("Another")}  
  
    public static void main(String args[]){  
        MyClass obj = new MyClass();  
        obj.print();  
        obj.show();  
        obj.anotherMethod();  
    }  
}
```

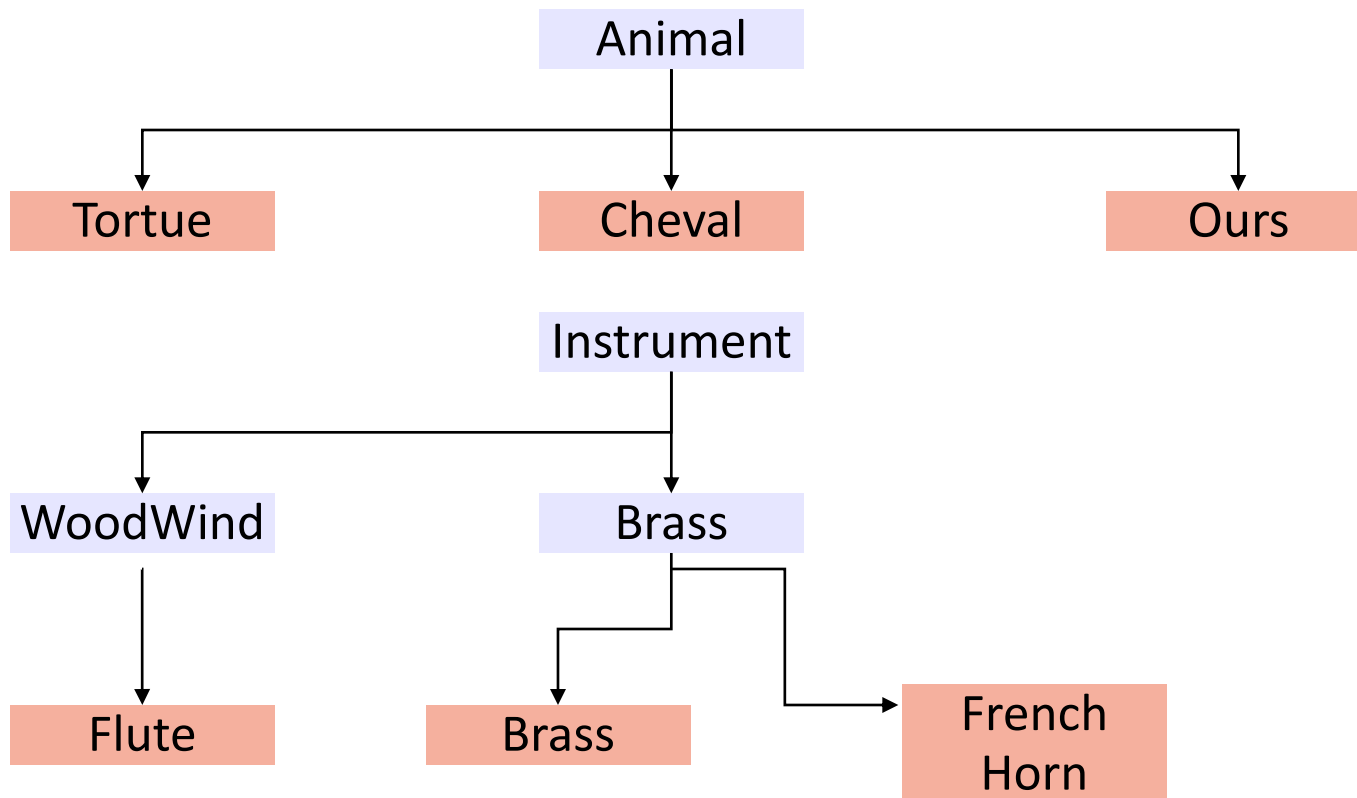
interface

Héritage multiple ?

```
public class MyDerivedClass extends MyClass implements Printable, Showable {  
    ...  
}
```


Classes abstraites

- Une classe abstraite est une classe non-instanciable.
- Une classe `Instrument` *ne représente pas une entité concrète. Un instrument n'existe pas, une Flute oui. Pareil pour Animal, c'est un concept.*



abstract

```
public abstract class Media {  
  
    ...  
}
```

- Classe abstraite : **abstract**
- Toujours avec des classes dérivées concrètes
- Une classe peut être déclarée abstraite mais contenir des méthodes implémentées.
- Une classe est également abstraite si elle n'est pas entièrement implémentée :
 - si une classe a (au moins) une méthode `abstract` alors cette classe est abstraite
 - ses sous-classes doivent terminer son implémentation.
- Si une sous classe d'une classe abstraite n'implémente pas toutes les méthodes "`abstract`" de sa classe mère, c'est aussi une classe abstraite.
- Une classe abstraite ne peut pas être instanciée (*new*).
- Une référence d'un type abstrait peut exister (*upcasting*).
- Une **interface** est une classe abstraite pure.

Example

```
abstract class Instrument {
    int i;
    public abstract void play();
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play() {
        System.out.println("
            Wind.play()");
    }
    public void adjust() {...}
}

class Brass extends Instrument {
    public void play() {
        System.out.println
            ("Brass.play()");
    }
    public void adjust() {...}
}
```

...

```
public class Music {
    static void tune(Instrument i) {
        // ...
        i.play();
    }
    static void tuneAll(
        Instrument[] e) {
        for(int i=0;i<e.length;i++)
            tune(e[i]);
    }
    public static void main
        (String[] args) {
        Instrument[] orchestra = new
        Instrument[2];
        int i = 0;
        orchestra[i++] = new Wind();
        orchestra[i++] = new Brass();
        tuneAll(orchestra);
    }
}
```

Abstract vs. Interface

- Une interface est, par défaut, une classe purement abstraite

abstract

```
public abstract class Instrument {  
    protected int volume;  
    public Instrument(){volume = 5;}  
    public abstract void play();  
  
    public adjust(int dv){volume+=dv;}  
}
```

```
public class Wind extends Instrument {  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
}
```

```
public abstract class Brass extends Instrument {  
    public Brass(){volume = 10;}  
}
```

Polymorphisme

- Upcasting + lien dynamique = polymorphisme
- Le terme polymorphisme est la caractéristique d'un élément pouvant prendre plusieurs formes : l'eau peut se trouver à l'état solide, liquide ou gazeux.
- En programmation Objet, polymorphisme est le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe tout en restant lui même.
- Le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.
- *"Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage"* Bruce Eckel "Thinking in JAVA"

Polymorphisme

Toujours binding dynamique

```
public class A {  
    public void m1(){  
        System.out( "m1 de A");  
    }  
}  
  
public class B extends A {  
    public void m1(){  
        System.out( "m1 de B");  
    }  
    public void m2(){  
        System.out( "m2 de B");  
    }  
}
```

```
A obj;  
  
double hasard = Math.random();  
  
if (hasard>0.5) obj = new A();  
else obj = new B(); //Upcasting  
  
obj.m1(); // lien dynamique  
  
// obj.m2() ne peut pas être  
résolu.  
}
```

8-CLASSES PARTICULIÈRE

Classes imbriquées

```
public class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
  
    class InnerClass {  
        ...  
    }  
}
```

- ✓ Ce sont des classes déclarées à l'intérieur d'autres classes.
- ✓ Ce sont des classes membres de la classe englobante (OuterClass).
- ✓ Elles peuvent être déclarées public, private ou protected.

Classes imbriquées (non statiques)

```
public class OuterClass {  
    private int a;  
    class InnerClass {  
  
        void method(){a=10;}  
  
    }  
}
```



Création d'objets :

```
public class Exemple {  
    public static void main(String args[]){  
  
        OuterClass oObj = new OuterClass();  
        OuterClass.InnerClass iObj = oObj.new InnerClass();  
  
    }  
}
```

Classes imbriquées (statiques)

```
public class OuterClass {  
    public int a;  
    static class StaticNestedClass {  
        void innerMethod(){a = 10;}  
    }  
}
```

INTERDIT

```
public class OuterClass {  
    public int a;  
    static class StaticNestedClass {  
        void innerMethod(){  
            OuterClass obj = new OuterClass();  
            obj.a = 10;  
        }  
    }  
}
```

OK

Classes imbriquées (statiques)

```
public class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
}
```

Création d'objets :

```
public class Exemple {  
    public static void main(String args[]){  
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();  
    }  
}
```

Classes anonymes

```
public interface Printable {  
    void print();  
}
```

- Une classe anonyme crée un objet d'une classe qui soit hérite (**extend**) une classe existante ou implémente (**implements**) une interface.

- La classe anonyme a accès aux variables de la classe englobante.

```
public class MyClass {  
    public void method(Printable p){  
        p.print();  
    }  
}  
  
public class ExempleAnonymousApp {  
    public static void main(String args[]){  
        MyClass mc = new MyClass();  
        mc.method(  
            new Printable(){  
                void print(){ System.out.println("Hello");}  
            }  
        )  
    }  
}
```

2.4 EXCEPTIONS

Qu'est ce qu'une exception

Une exception est un événement qui a lieu en cours d'exécution d'un programme et qui perturbe son déroulement

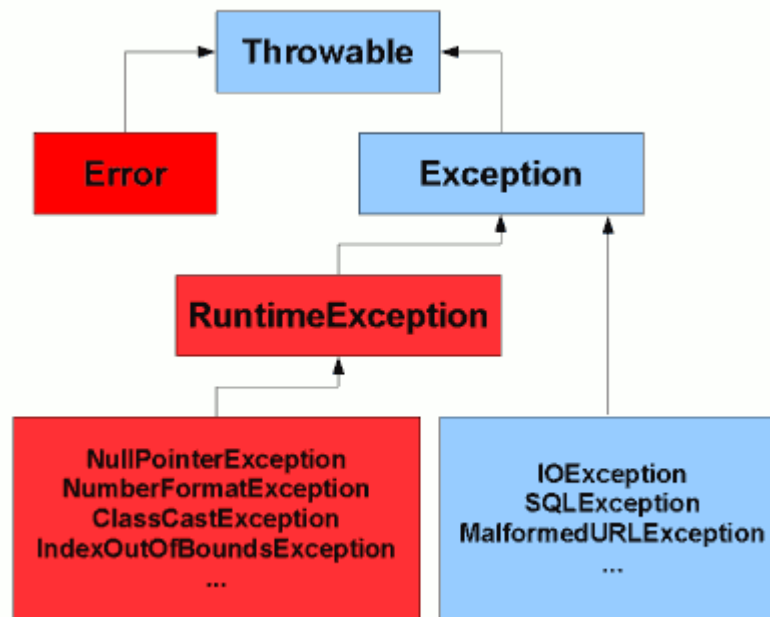
Exemple : Division par zéro, échec de conversion, erreur d'entrées-sorties

```
public class ExceptionExample {  
    public static void main(String[] args) {  
        String s = "bonjour";  
        System.out.println("la 8ème lettre est : "  
                           + s.charAt(8));  
    }  
}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:  
    String index out of range: 8  
at java.lang.String.charAt(String.java:558)  
at ExceptionExample.main(ExceptionExample.java:6)
```

Hiérarchie

Les exceptions sont des classes java dont la classe mère est Throwable



En bleu : Exceptions dont la gestion est obligatoire

En rouge : Exceptions dont la gestion est facultative

Attraper une exception

```
try {  
    // instructions qui peuvent provoquer une exception  
}  
catch (Exception e) {  
    // traitement en cas d'exception  
}  
finally {  
    // traitement dans tous les cas  
}
```

OPTIONNEL

Attention : Les instructions dans la clause *catch* ou *finally* ne sont pas surveillées par le mécanisme

Que faire d'une exception ?

Quitter le programme et reporter l'exception sur la sortie standard

```
char c;  
try {  
    c = s.charAt(i);  
}  
catch(StringIndexOutOfBoundsException e) {  
    e.printStackTrace();  
}
```

"Corriger" l'erreur

```
char c;  
try {  
    c = s.charAt(i);  
}  
catch(StringIndexOutOfBoundsException e) {  
    if(s.length() > 0)  
        c = s.charAt(0);  
}
```

Un try et plusieurs catch

Traiter plusieurs exceptions "d'un coup" (plusieurs catch)

```
public class ReadInt {  
    public static void main(String[] args) {  
  
        try {  
            System.out.println( "Entier : "+ Integer.parseInt(args[0]));  
  
            int[] tab = new int[10];  
            System.out.println( "Booléen : " + tab[12]==tab[0]);  
  
        } catch( NumberFormatException e1) {  
            System.err.println( "Ce n'est pas un entier !");  
        } catch( Exception e2 ) {  
            e2.printStackTrace();  
        } finally {  
            System.err.println( "Fin du try/catch");  
        }  
    }  
}
```

Ordre des catch

Attention à l'**ordre** des catch en fonction de l'**héritage** !

```
public class ReadInt {  
    public static void main(String[] args) {  
  
        try {  
            System.out.println( "Entier : "+ Integer.parseInt(args[0]));  
  
            int[] tab = new int[10];  
            System.out.println( "Booléen : " + tab[12]==tab[0]);  
  
        } catch( Exception e2 ) { // ATTENTION, CE CATCH ATTRAPE NumberFormatException  
            e2.printStackTrace();  
        } catch( NumberFormatException e1) {  
            System.err.println( "Ce n'est pas un entier !");  
        } finally {  
            System.err.println( "Fin du try/catch");  
        }  
    }  
}
```

Faire mes exceptions

Définir des sous-classes de Exception,
RuntimeException ou Error.

La plupart du temps, il s'agit de cas qui doivent
être anticipés : sous-classes d'Exception.

Faire mes exceptions : exemple

```
public class PersonNotFoundException extends Exception {  
  
    public PersonNotFoundException () {  
        super();  
    }  
  
    public PersonNotFoundException(String name) {  
        super( "Cannot find "+name);  
    }  
  
}
```

Jeter une exception

Utiliser l'instruction **throw** et le mot-clé **throws**.

```
import java.util.LinkedList;
public class ListOfPersons extends LinkedList {

    ...

    public int indexOf(String namePerson) throws PersonNotFoundException {
        int index = super.indexOf(namePerson);
        if(index == -1)
            throw new PersonNotFoundException(namePerson);
        else
            return index;
    }
}
```

Une méthode doit indiquer dans sa signature les classes d'exception qu'elle jette

Propager une exception

Une méthode peut propager une exception pour que le traitement de l'erreur soit fait par celui qui l'appelle : mot-clé **throws**.

```
import java.util.LinkedList;
public class ListOfPersons extends LinkedList {

    ...

    public int indexBefore(String namePerson)
        throws PersonNotFoundException {
        return indexOf(namePerson)-1;
    }
}
```


Exception : exemple résumé

```
public class Thermometre {  
    private int temperature = 0;  
  
    public Thermometre() { }  
  
    public chauffe(int degre) throws MaladeException {  
        if((temperature + degre) > 40)  
            throw new MaladeException();  
        temperature += degre;  
    }  
  
    public baisse(int degre) {  
        temperature -= degre;  
    }  
    ...  
}
```

Exception : exemple résumé

```
public class Thermometre {  
    ...  
    class MaladeException extends Exception {  
        MaladeException() {  
            super();  
        }  
    }  
}  
  
public static void main(String[] args) {  
    Thermometre th = new Thermometre();  
    try {  
        th.chauffe(38);  
    } catch(MaladeException e) {  
        // Faire quelque chose  
    }  
}
```