(2): le principe de l'héritage

- Une classe B hérite d'une classe A si tout objet de B est objet de la classe mère A
-

Exemple

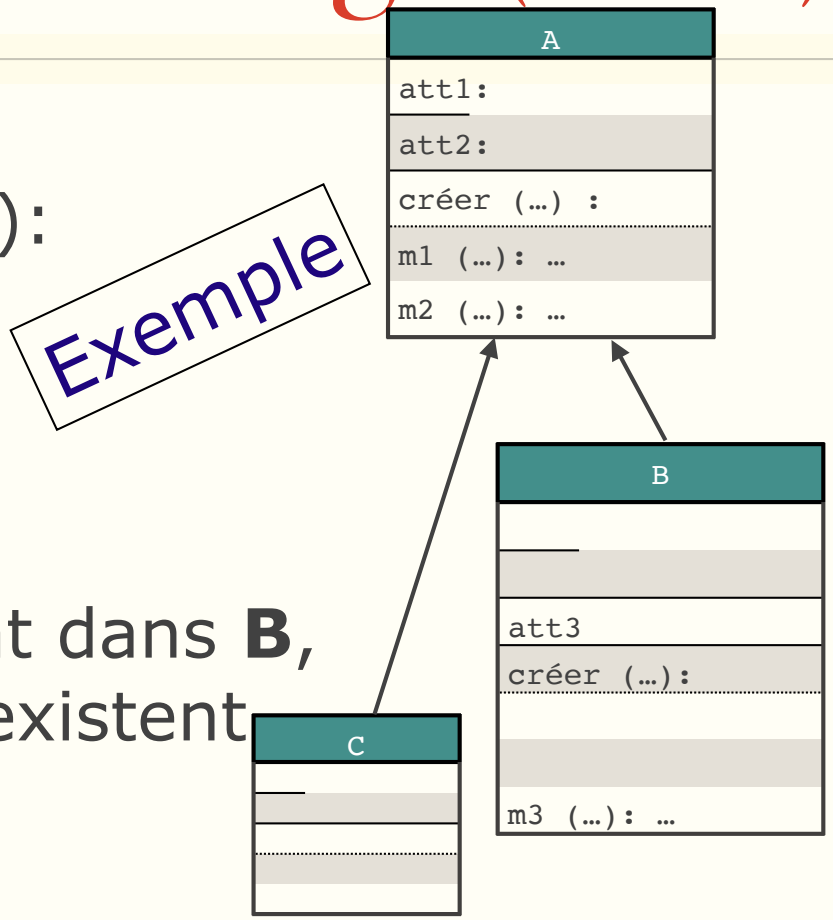
La classe Rectangle hérite de la classe Quadrilatère

Rmq. pourrait aussi hériter de Parallélogramme, mais…

- Réutilisation et spécialisation :
  - Réutiliser ce qui existe
  - Modifier ce qui est différent
  - Rajouter ce qui manque

(2): Le principe de l'héritage (suite)

- Si la classe Fille **B** hérite de la classe Mère **A** (de même que **C**):



- tous les attributs de **A** existent dans **B**,
- les attributs définis dans **B** n'existent pas dans **A**,

- toutes les méthodes de **A** existent dans **B**,
- les méthodes de **A** peuvent être redéfinies dans **B,**
- les méthodes (re)définies dans **B** n'existent pas dans **A**.

(2): Héritage (suite) : nouvelle définition, redéfinition et surcharge

- Une sous-classe peut donc **définir** des **nouvelles** méthodes :
  - pour des méthodes qui n'existaient pas dans la classe mère

- Une sous-classe peut aussi **redéfinir** des méthodes dont elle hérite :
  - Implémentations spécialisées pour la classe fille
    - s'appuyant éventuellement sur le code existant
  - Condition de redéfinition : une méthode à signature identique, i.e.:
    - Même nom
    - Même type de retour
    - Même liste et type d'arguments
  - La méthode redéfinie est appelée en priorité par rapport à une instance de la classe fille
    -> liaison dynamique (voir principe (4))

- Une classe peut **surcharger** des méthodes :
  - Plusieurs implémentations pour une même méthode
  - se différentient par le nombre et le type des paramètres

```
class Point {
            int x, y;
            Point(int x, int y) {  . . .  }
            double distance(int x, int y) {. . .}
            double distance(Point p) { . . . }
}
```

(2): Héritage (suite) : nouvelle définition, redéfinition et surcharge

- Une sous-classe peut donc **définir** des **nouvelles** méthodes :
  - pour des méthodes qui n'existaient pas dans la classe mère

- Une sous-classe peut aussi **redéfinir** des méthodes dont elle hérite :
  - Implémentations spécialisées pour la classe fille
    - s'appuyant éventuellement sur le code existant
  - Condition de redéfinition : une méthode à signature identique, i.e.:
    - Même nom
    - Même type de retour
    - Même liste et type d'arguments
  - La méthode redéfinie est appelée en priorité par rapport à une instance de la classe fille
    -> liaison dynamique (voir principe (4))

- Une classe peut **surcharger** des méthodes :
  - Plusieurs implémentations pour une même méthode
  - se différentient par le nombre et le type des paramètres

```
class Point {
            int x, y;
            Point(int x, int y) {  . . .  }
            double distance(int x, int y) {. . .}
            double distance(Point p) { . . . }
}
```

(2): Héritage (suite) : nouvelle définition, redéfinition et surcharge

- Une sous-classe peut donc **définir** des **nouvelles** méthodes :
  - pour des méthodes qui n'existaient pas dans la classe mère
- Une sous-classe peut aussi **redéfinir** des méthodes dont elle hérite :
  - Implémentations spécialisées pour la classe fille
    - s'appuyant éventuellement sur le code existant
  - Condition de redéfinition : une méthode à signature identique, i.e.:
    - Même nom
    - Même type de retour
    - Même liste et type d'arguments
  - La méthode redéfinie est appelée en priorité par rapport à une instance de la classe fille

    -> liaison dynamique (voir principe (4))

- Une classe peut **surcharger** des méthodes :
  - Plusieurs implémentations pour une même méthode
  - se différentient par le nombre et le type des paramètres

Remarque: la surcharge n'est pas liée à l'héritage mais c'est mieux de la présenter ici

```
class Point {
            int x, y;
            Point(int x, int y) {  . . .  }
            double distance(int x, int y) {. . .}
            double distance(Point p) { . . . }
}
```

Quelques exercices de travaux pratiques avant d'aller plus loin

Travail demandé :

- …
- …
- …

Travail pratique n°4 : héritage en Java
Segments colorés

```
public class Segment {
    ... ;
    public Segment(..., ... ) {
        ... // à compléter pour créer un segment
    }
    public Segment(...) {
        ... // à compléter pour créer en copiant un autre segment
    }
    public ... translater(...) {
        ...  // à compléter }
    public void afficher() {... }
        ... // à compléter }
}
```

Travail demandé :

- en s'inspirant de la classe **`Segment`** (ci-dessus), créer la classe **`SegmentColore`** des segments colorés (en considérant qu'il existe une classe **`Color`**)

- *définir les **constructeurs** de segments colorés*
- *ajouter une méthode pour **connaître la couleur** d'un segment coloré*
- *ajouter une méthode pour **afficher** les coordonnées des extrémités du segment*

Travail pratique n°4:
héritage en Java
Segments colorés

```
public class SegmentColoré ... {
    ... ;
    public SegmentColoré(..., ... ) {
        ... // à compléter pour créer un segment coloré
    }
    public SegmentColoré(...) {
        ... // à compléter pour créer en copiant un autre segment
    }
    public ... translater(...) {
        ...  // à compléter }
    public void afficher() {... }
        ... // à compléter }
}
```

Travail demandé :

- en s'inspirant de la classe **`Segment`**, créer la classe **`SegmentColore`** des segments colorés (en considérant qu'il existe une classe **`Color`**)

- *définir les **constructeurs** de segments colorés*
- *ajouter une méthode pour **connaître la couleur** d'un segment coloré*
- *ajouter une méthode pour **afficher** les coordonnées des extrémités du segment*

Travail pratique n°5:
héritage en Java
Couples

```
public class Couple<T> ... {

    public Couple(...) {

        ...

    }

    public String toString() {

        ...

    }

}
```

Travail demandé :

- concevoir une classe **`Couple`**, tel que tout élément de cette nouvelle classe est élément spécifique de la classe **`Vector<T>`**

- *définir les **constructeurs** de couple*
- *ajouter une méthode pour **appliquer un traitement** aux deux éléments du couple*
- *redéfinir la méthode **`toString()`** pour **afficher** les éléments du couple*

Travail pratique n°5: héritage en Java

SOLUTION

```
public class Couple<T> extends Vector<T> {
    public Couple(T a, T b) {
        super(); this.add(a); this.add(b);
    }
     public String toString() {
         String s = new String();
         for(Object obj : this) {
            s+=obj.toString()+" --- ";
        }
        return(s);
    }
}
```

Travail demandé :

- concevoir une classe **`Couple`**, tel que tout élément de cette nouvelle classe est élément spécifique de la classe **`Vector<T>`**

- *définir les **constructeurs** de couple*
- *ajouter une méthode pour **appliquer un traitement** aux deux éléments du couple*
- *redéfinir la méthode **`toString()`** pour **afficher** les éléments du couple*

Travail pratique n°6: héritage en Java

```
public class Segment ... {

    public Segment(Point p1, Point p2) {
        ...
    }

    public String toString() {
        ...
    }
}
```

```
class Point{
  protected int x;
  protected int y;
  public Point(int abs,int ord) {
      x = abs; y = ord;
    }
    public String toString() {
        return ("abs = "+x+" , ord = "+y);
    }
```

Travail demandé :

- concevoir maintenant la classe **`Segment`** comme un couple de **`Point`**
- *définir les **constructeurs** de segment*
- *redéfinir la méthode toString()*

Travail pratique n°6: héritage en Java

SOLUTION

```
public class Segment extends Couple<Point> {

    public Segment(Point p1, Point p2) {
        super(p1,p2);
    }

    public String toString() {
        return("Segment = "+super.toString());
    }
}
```

```
class Point{
  protected int x;
  protected int y;
  public Point(int abs,int ord) {
      x = abs; y = ord;
    }
    public String toString() {
        return ("abs = "+x+" , ord = "+y);
    }
```

Travail demandé :

- concevoir maintenant la classe **`Segment`** comme un couple de **`Point`**
- *définir les **constructeurs** de segment*
- *redéfinir la méthode toString()*