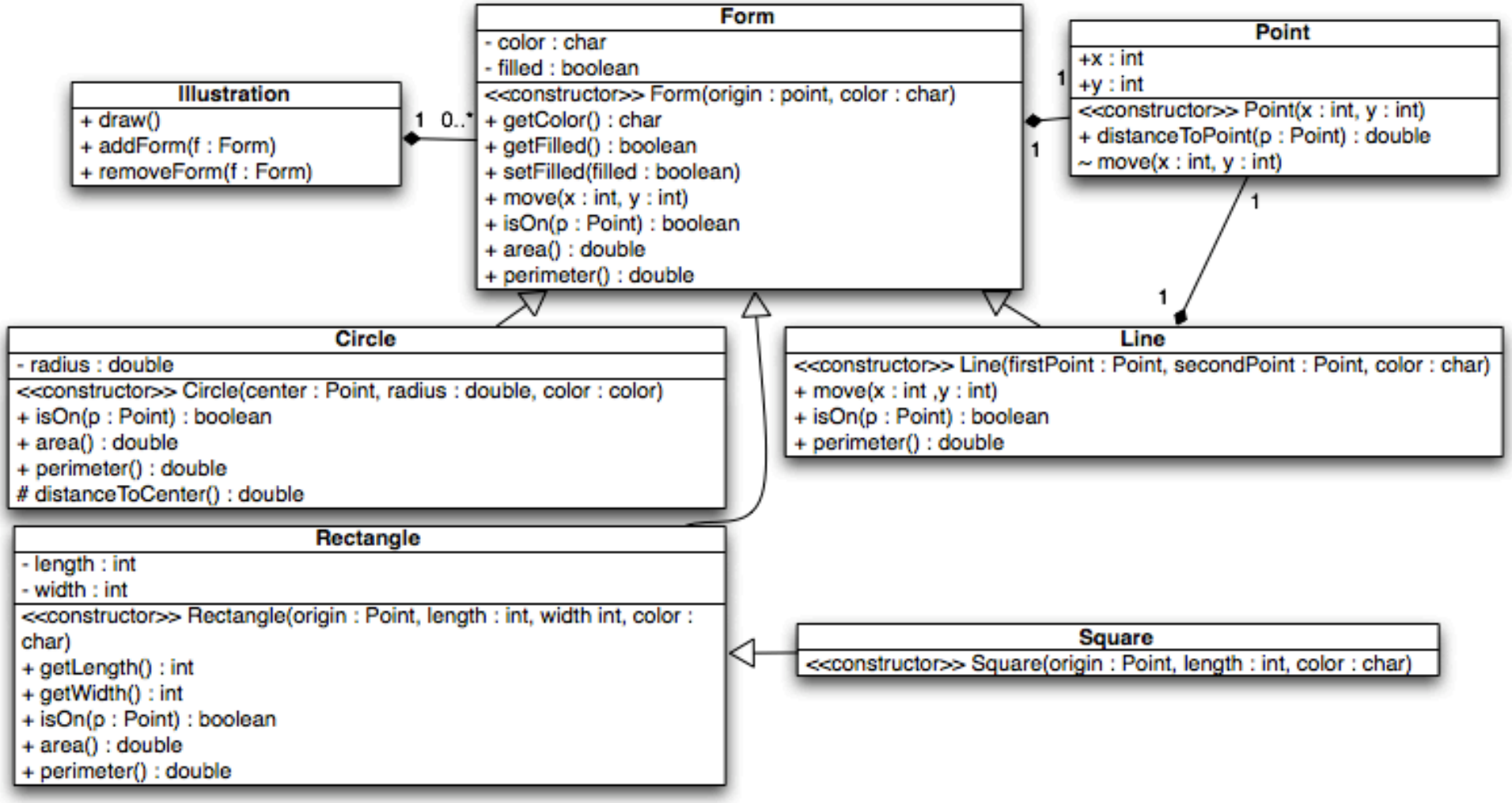
ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H-301
Programmation orientée objet

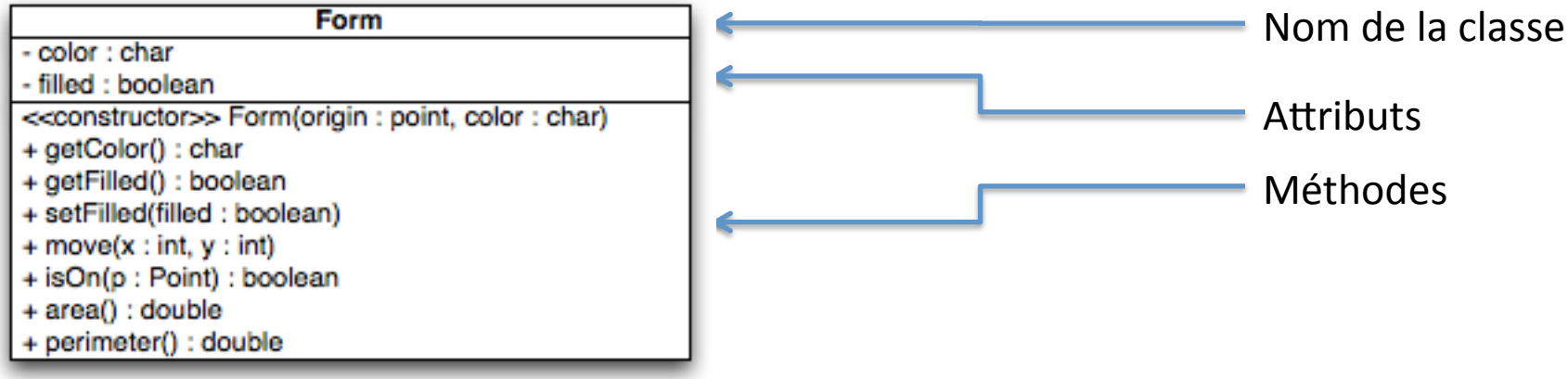
TP3
- Diagramme de classes et associations
- Classes abstraites et polymorphisme

F. Servais & B. Verhaegen

Diagramme de classes UML

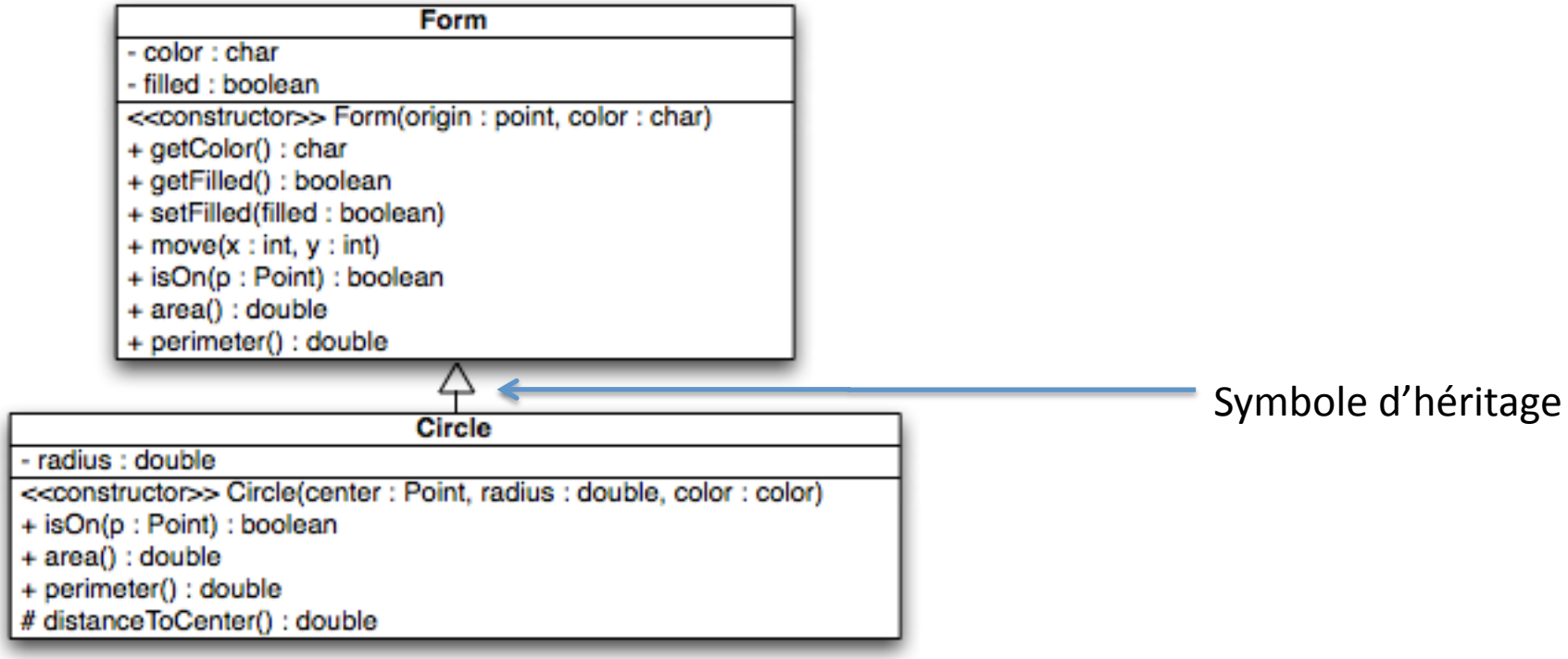


Classes, méthodes et attributs



- Encapsulation :
  - `+ : public`
  - `- : private`
  - `# : protected`
  - `~ : package-private`
- Les types sont précisés à l'aide d'un `" : "`
- Le constructeur est précédé par `<<constructor>>`
- Les membres statiques sont soulignés

Héritage



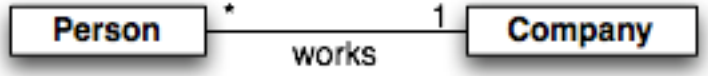
- Les méthodes redéfinies sont réécrites dans la classe fille ainsi que les méthodes et attributs supplémentaires.

Association, composition, agrégation

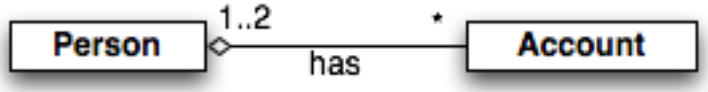
- Une **association** est une relation sémantique entre des classes qui définit un ensemble de liens.
  - Un homme est associé à sa femme
- Une **agrégation** est une association dans laquelle il y a un lien d'appartenance entre les deux objets associés (contenant/contenu, possession, …).
  - Un homme possède un compte en banque
- Une **composition** (ou agrégation forte) est une agrégation dans laquelle la disparition du composite entraine la disparition des composants.
  - Si un arbre meurt, ses feuilles ne servent plus à rien (on ne peut pas les mettre sur un autre arbre, au contraire de roues sur une voiture)
- Il s'agit surtout d'une différence sémantique qui impliquera des changements dans votre implémentation au niveau du cycle de vie des objets.

Association, composition, agrégation

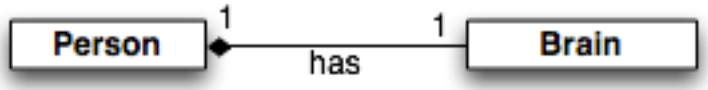
Association : Une personne travaille pour une et une seule compagnie



Agrégation : Une personne possède entre 0 et n comptes



Composition : Une personne a un et un seul cerveau



- Cardinalités :
  - **, n, m..*,* où *n>0* et *m>=0*
  - Par défaut, la cardinalité est *un*.
  - Dans une composition, la cardinalité du côté de l'agrégat ne peut être que 1 ou 0..1
- Le nom de l'association est facultatif

Associations multiples en Java

- En Java, les associations multiples peuvent par exemple s'implémenter dans des collections comme ArrayList, HashMap, …
- http://java.sun.com/docs/books/tutorial/collections/

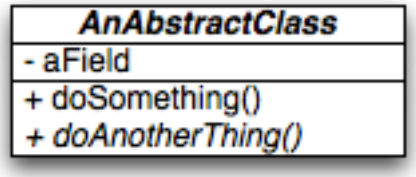
- Exemple d'utilisation d'ArrayList :

```
import java.util.*;
…
ArrayList<Person> myList = new ArrayList<Person>();
myList.add(new Person("Jean"));
myList.add(new Person("Hugues"));

for(Person p : myList) {
  System.out.println(p);
}
```

Classes abstraites

- Une **classe abstraite** ne peut être instanciée mais peut être *sous-classée*. Elle peut contenir ou non des méthodes abstraites.
- Uné **méthode abstraite** est une méthode déclarée mais non implémentée.
- Si une classe inclut des méthodes abstraites elle doit être abstraite.
- Une classe qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de la classe parente. Sinon elle doit être déclarée abstraite.




```
public abstract class AnAbstractClass{
  private int aFied;
  ...
  public void doSomething(){ ... }
  public abstract void doAnotherThing();
}
```

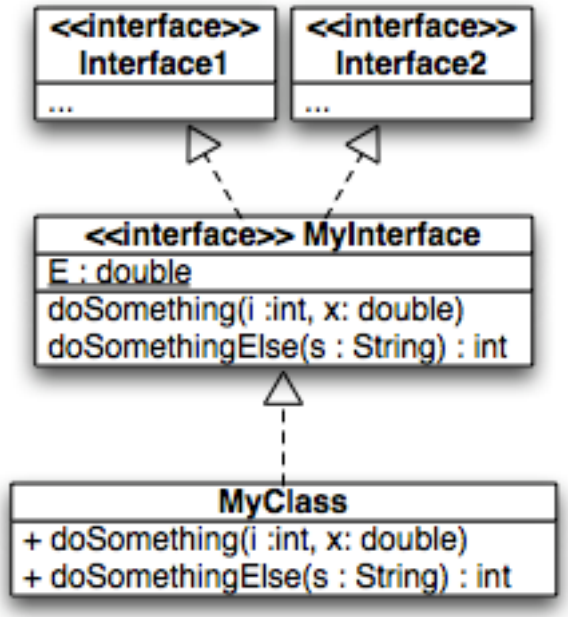
Interfaces

- Une **interface** est un cas particulier d'une classe abstraite qui ne peut contenir que des constantes et **signatures de méthodes.**
- Il n'y a pas de niveau de protection à préciser :
  - Les constantes sont implicitement `public static` et `final`
  - Les méthodes sont implicitement `public`
- Elle ne peut être qu'***implémentée*** par des classes ou ***étendue*** par d'autres interfaces.
- En Java, une classe ne peut hériter que d'une autre classe mais peut implémenter plusieurs interfaces. De ce fait, un objet peut avoir **plusieurs types** : le type de sa classe et les types des interfaces que cette classe implémente.
- http://java.sun.com/docs/books/tutorial/java/landl/createinterface.html

Interfaces en Java et en UML

```
public interface MyInterface extends Interface1, Interface2
{
  double E = 2.718282;
  void doSomething (int i, double x);
  int doSomethingElse(String s);
}

public class MyClass implements MyInterface {
  public void doSomething(int i ,double x) {
    ...
  }
  public int doSomethingElse(String s){
    ...
  }
}
...
MyInterface m = new MyClass(); //Interface utilisée comme type
m.doSomething(4,2.1);
```



Polymorphisme

- Le **polymorphisme** est un **concept OO** dont l'idée est d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.
- Vous l'avez utilisé implicitement dans l'exercice précédent (TP2 2e partie) lorsque vous avez parcouru les formes d'une illustration :

```
ArrayList<Form> formList = new ArrayList<Form>();
formList.add(new Square(aPoint, 10));
formList.add(new Line(aPoint, anotherPoint));
...
foreach(Form f : formList){
  f.doSomething();
}
```

Transtypage

- En Java, un objet peut être *casté* en un autre objet (*typecasting* ou transtypage) :
  - soit en un objet d'une sous classe (*downcasting*)
  - soit en un objet d'une super-classe (*upcasting*)
- Une erreur de compilation aura lieu dans le cas d'un transtypage impossible (entre objets non liés par héritage, comme par exemple deux frères).
- Une exception sera déclenchée lors de l'exécution si l'objet *casté* n'est pas compatible avec le type dans lequel il a été *casté*.
- L'*upcasting* est automatique. Le *downcasting* nécessite un opérateur :
  - `Person p = new Assistant(…);`
  - `Assistant a = (Assistant)p;`

Exceptions et redéfinition

- Une méthode qui redéfini une méthode d'une super-classe peut uniquement déclencher les exceptions prévues dans la méthode de la super-classe.

- Elle ne peut pas déclencher d'exception d'un type non compatible.

Construction d'objet

```
public class A {
  public A(){
    …
  }
}
public class B extends A {
 public B(){
    …
  }
}
public class C extends B {
 public C(){
    …
  }
}
```

Lors de la construction d'un objet de la classe C :

- le constructeur de C commence par appeler le construteur de B
- le constructeur de B commence par appeler le constructeur de A
- le constructeur de A appelle le constructeur d'Object
- le constructeur d'Object s'exécute
- le constructeur de A se termine
- le constructeur de B se termine
- le constructeur de C se termine

Garbage collector

- En Java, quand un objet n'est plus référencé, il est détruit par un ramasse-miettes ou *garbage collector*.
- Il ne faut donc pas utiliser l'instruction `delete` pour libérer la mémoire comme en C++.
- Avant la **destruction** d'un objet, sa méthode `void finalize()` est appellée.
- Cette méthode est souvent utilisée quand on utilise des ressources « non-Java » comme des fichiers ou un réseau et qu'il faut libérer ces ressources.
- Lorsque le *garbage collector* détruit C :
  - le `finalize` de C (ou le plus proche de C en montant la hiérarchie) est appellé
  - la mémoire est libérée

Garbage collector

- Une bonne pratique est d'utiliser *try-catch-finaly* dans les `finalize` pour être certain de ne pas oublier de fermer les ressources réservées par les super-classes.

```
protected void finalize() throws Throwable {
  try {
    close(); // close open files
  } finally {
    super.finalize();
  }
}
```

- Si une exception est déclenchée dans un `finalize`, la finalisation est arretée mais l'exception est ignorée.
- `finalize()` n'est jamais appellé qu'une fois sur un objet.

Question

- Sachant comment fonctionne le *garbage collector*, quelle différence voyez vous entre l'implémentation d'une **agrégation** et celle d'une **composition** ?