ISEN ALL IS DIGITAL! OUEST | yncréa

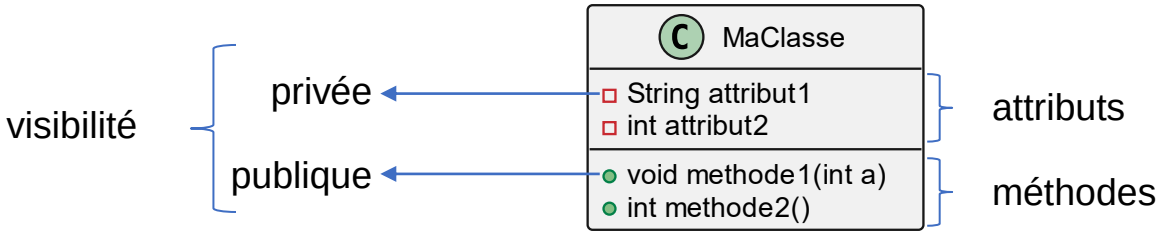
Java

L’héritage

Rappels sur l’héritage

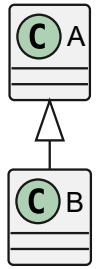
Préliminaires : UML

- Dans ce cours, nous allons utiliser la notation UML (Unified Modeling Language) pour représenter certains concepts objets de manière graphique et indépendante à tout langage.
- En UML, une classe se représente de la manière suivante :



L’héritage d’un point de vue conceptuel

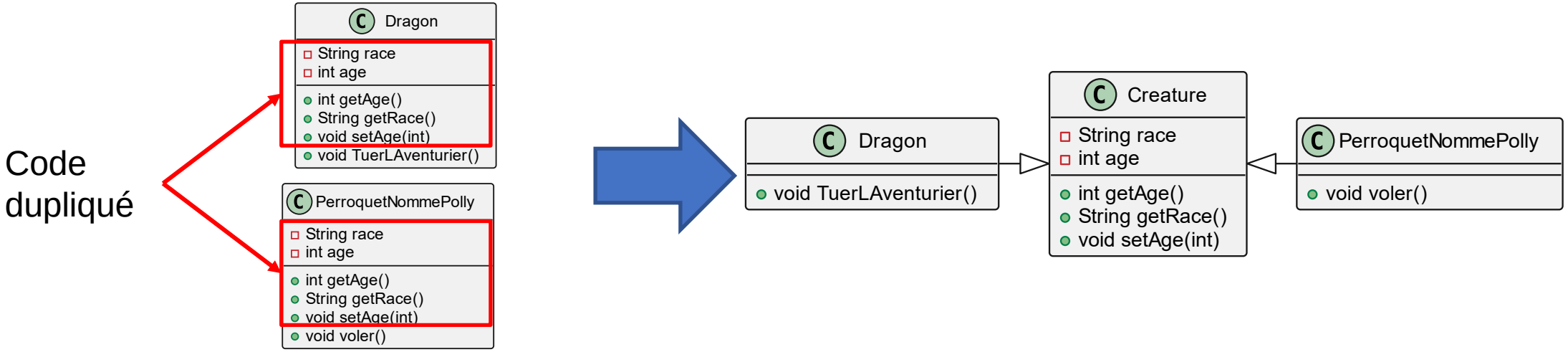
- L’héritage permet de modéliser une relation du type “est un(e)”.
- Par exemple :
  - Un chien est un animal
  - Un cercle est une forme
- En UML, la relation « B hérite de A » est illustrée par :



- On dit que :
  - A est une classe mère (ou super classe) de B
  - B est une classe fille (ou sous classe) de A

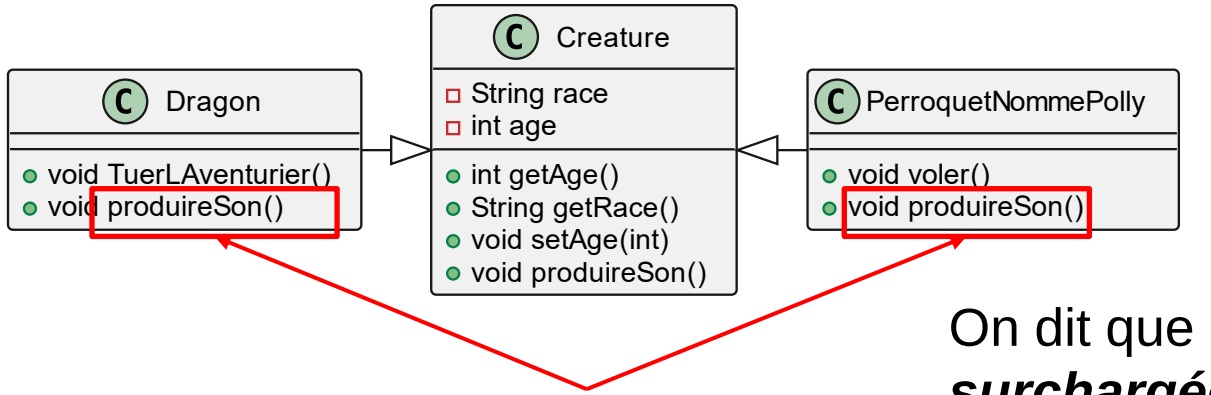
L'héritage d'un point de vue développement logiciel

- L'héritage permet de factoriser du code : tous les attributs et méthodes partagés par plusieurs concepts sont placés dans leur parent commun.
- On ne (re)définit que les parties spécifiques dans les classes filles, les parties héritées étant réutilisables.



Le polymorphisme d'héritage

- L'héritage permet aussi de définir des méthodes qui seront présentes dans des sous classes mais avec un comportement différent
- Ce mécanisme est appelé ***polymorphisme d'héritage***



Polymorphisme de classe

Merci à Dorian Tarin pour la photo de la carte !

On dit que la méthode produireSon() est ***surchargée*** dans les classes Chien et Oiseau

L’héritage en Java

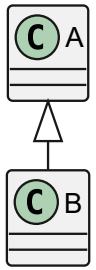
Tout est Object

- Object est la classe au sommet de la hiérarchie des classes Java
  - Toute classe Java hérite (in)directement de Object
- Object définit des opérations élémentaires sur des objets, notamment une implémentation basique des opérations suivantes :

Opération	Méthode
Test d'égalité avec un autre objet	`boolean equals (Object other)`
Représentation sous forme de chaîne de caractères	`String toString()`
Recopie de l'objet	`Object clone()`

- Ces méthodes sont très souvent surchargées

Relation d'héritage en Java

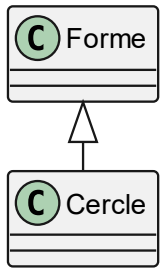



```
public class B extends A
{
        ...
}
```

- ⚠ On ne peut hériter que d'une seule classe (pas d'héritage multiple en Java)
  - Mais il existe des moyens de contourner cette limitation, notamment implémenter plusieurs interfaces (voir cours dédié).

Exemple : un cercle est une forme

UML



Java

```java
public class Forme
{
	public void afficher(){
		System.out.println("Je suis en forme");
	}
}
```

```java
public class Cercle extends Forme
{
	public void afficher(){
		System.out.println("Je tourne en rond");
	}
}
```

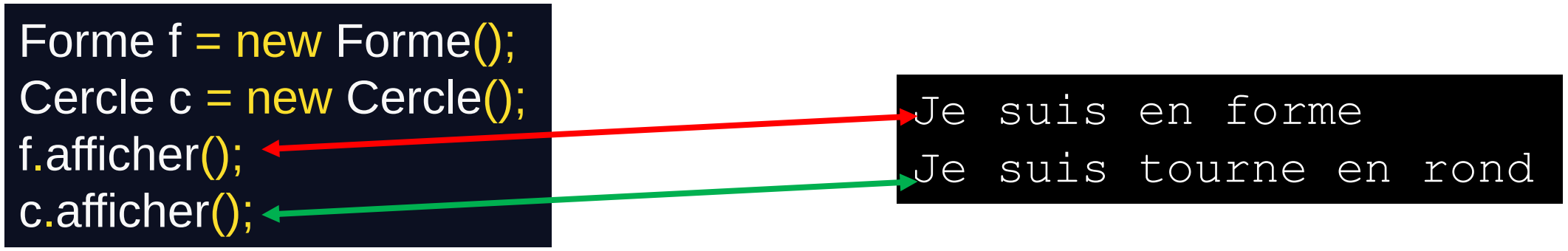
Exemple : un cercle est une forme

Utilisation

```
Forme f = new Forme();
Cercle c = new Cercle();
f.afficher();
c.afficher();
```

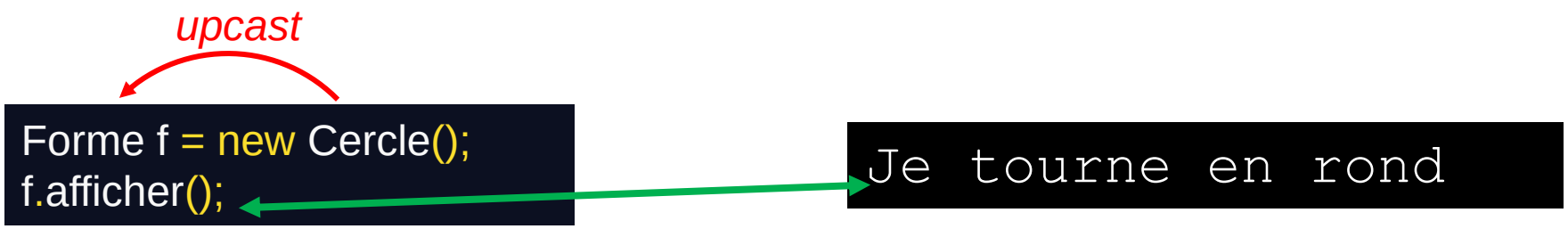
Sortie standard

```
Je suis en forme
Je suis tourne en rond
```



Upcast

- Comme un Cercle est un cas particulier de Forme, on peut affecter une instance de Cercle à une variable de type Forme
- Cette opération est appelée transtypage ascendant *upcast*
- Au moment de l'appel à une méthode, la version surchargée sera appelée si elle existe.



Accès aux membres hérités

Accès protégé

- Une sous classe peut accéder à tous les membres publics ce ses super classes (comme si c'était les siens).
- Mais le principe d'encapsulation interdit l'accès aux membres hérités privés, ce qui peut rendre leur manipulation fastidieuse voire impossible dans les surcharge.
- Il existe donc une visibilité intermédiaire, appelée ***protegée*** , qui donne accès privilégié aux sous classes
- Les membres protégés sont publics du point de vue des sous classes, et privés du point de vue des autres classes.

Accès protégé

Visibilité protégée

```
public class Forme
{
        protected Point centre;
        ...
}
```

Accès direct à centre possible dans la surcharge.

```
public class Cercle extends Forme
{
        public void afficher(){
                System.out.println("Je suis un cercle de centre " +
centre.toString());
        }
}
```

Accès protégé

Visibilité protégée

```
public class Forme
{
        protected Point centre;
        ...
}
```

Appel implicite à toString()

```
public class Cercle extends Forme
{
        public void afficher(){
                System.out.println("Je suis un cercle de centre " +
centre);
        }
}
```

Le désignateur super

- Le désignateur super permet de faire référence à la super classe.
- Il permet d'appeler des méthodes héritées.
- Cas d'utilisation courants :
  - Compléter une opération plutôt que de la redéfinir totalement (et ainsi réutiliser ce qui a déjà été implémenté)
  - Déléguer l'initialisation des membres hérités en appelant le constructeur de la super classe.

Appel aux méthodes héritées

```
public class Forme {
        protected Point centre;
        ...
        public void afficher(){
                System.out.println("Je suis une forme de centre " + centre);
        }
}
```

```
public class Cercle extends Forme {
        private float rayon;
        ...
        public void afficher(){
                super.afficher();
                System.out.println("Je suis un cercle de rayon " + this.rayon);
        }
}
```

Appel aux méthodes héritées

```
public class Cercle extends Forme {
        private float rayon;
        ...
        public void afficher(){
                super.afficher();
                System.out.println("Je
suis un cercle de rayon " + this.rayon);
        }
}
```

Utilisation

```
Point p = new Point(100, 200);
Cercle c = new Cercle(p, 10);
c.afficher();
```

Sortie standard

```
Je suis une forme de centre (100,200)
Je suis un cercle de rayon 10
```

Cas particulier du constructeur

- Chaque classe est responsable de l'intégrité de ses attributs.
- Le constructeur d'une sous classe doit donc uniquement initialiser ses attributs spécifiques, et appeler le constructeur de la super-classe pour initialiser les attributs hérités.
- Autrement dit, l'initialisation des membres hérités est délégué aux super-classes.

Exemple

Délégation de l'initialisation des attributs hérités à la super-classe

```
public class Forme
{
        protected Point centre;
        public Forme(Point centre){
                this.setCentre(centre)
        }
}
```

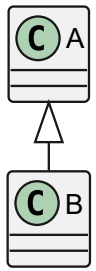
Initialisation des attributs propres

```
public class Cercle extends Forme
{
private float rayon;
        public Cercle(Point centre, float rayon){
                super(centre);
                this.setRayon(rayon);
        }
}
```

Quelques compléments*

L’opérateur instanceof

- L’opérateur instanceof permet de tester le type d’une instance i instanceof C renvoie true si i est, directement ou indirectement, une instance de la classe C
- Exemples :




```
A a = new A();
System.out.println(a instanceof A);
System.out.println(a instanceof B);
```

```
true
false
```

```
A a = new B();
System.out.println(a instanceof A);
System.out.println(a instanceof B);
```

```
true
true
```

L'annotation @Override

- Une annotation est une information destinée au compilateur Java
- L'annotation @Override permet de préciser qu'une méthode est une surcharge
- Exemple :

```
public class Cercle extends Forme
{
        @Override
        public void afficher(){...}
}
```

- Avantages :
  - Améliore la lisibilité du code : la surcharge est explicite
  - Evite les erreurs : une erreur se produira à la compilation s'il y a incohérence entre la définition de la méthode héritée et surchargée