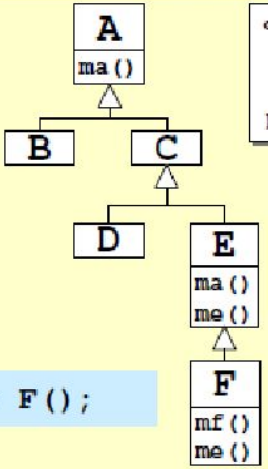
EXERCICES:

EXEMPLE TIRÉ DU COURS DU PROFESSEUR PHILIPPE GENOUD

Upcasting/Downcasting




```
class A {
    public void ma() {
        System.out.println("methode ma définie dans A");
    }
}
```

```
class E extends C {
    public void ma() {
        System.out.println("methode ma redéfinie dans E");
    }
    public void me() {
        System.out.println("methode me définie dans E");
    }
}
```

```
class F extends E {
    public void mf() {
        System.out.println("methode mf définie dans f");
    }
    public void me() {
        System.out.println("methode me redéfinie dans F");
    }
}
```

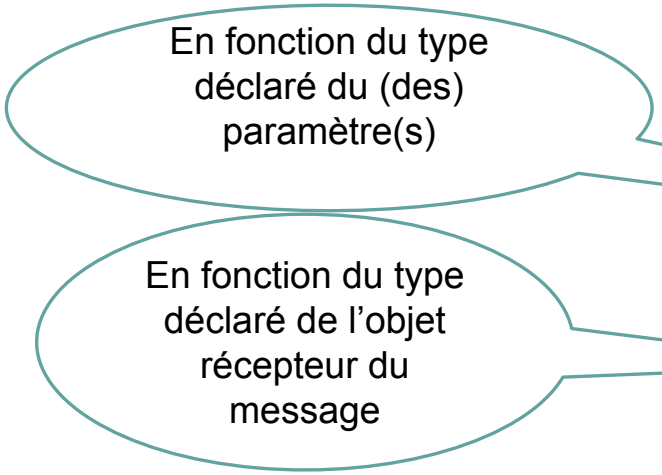
```
C c = new F();
```

	compilation	exécution
`c.ma();`	La classe C hérite d'une méthode ma	→ méthode ma définie dans E
`c.mf();`	*Cannot find symbol : metod mf()* Pas de méthode mf() définie au niveau de la classe C	
`B b = c;`	*Incompatible types* Un C n'est pas un B	
`E e = c;`	*Incompatible types* Un C n'est pas forcément un E	
`E e = (E)c;` `e.me();`	Transtypage (Dowcasting), le compilateur ne fait pas de vérification La classe E définit bien une méthode me	→ méthode me définie dans F
`D d = (D) c;`	Transtypage (Dowcasting), le compilateur ne fait pas de vérification	*ClassCastException* Un F n'est pas un D

RAPPEL: SURCHARGE ET POLYMORPHISME

```
public class Personne {
public void methode1(Personne e){
System.out.println("methode1 de personne");

}
```

```
public class Employe extends Personne {
public void methode1(Employe e){
System.out.println("methode1 de Employe");

}

public static void main(String[] args) {
Personne p=new Employe();
Employe e= new Employe();


e.methode1(p);

p.methode1(new Employe());

} }
```



choix de la méthode à exécuter est effectué **statistiquement à la compilation en fonction des types déclarés : Sélection Statique**

methode1 de personne
methode1 de personne

A PROPOS DE EQUALS

EXEMPLE TIRÉ DU COURS DU PROFESSEUR PHILIPPE GENOUD

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o;
 }
 ...
}
```

```
public class Point {

      private double x;
      private double y;

      ...

}
```

De manière générale, il vaut mieux éviter de surcharger des méthodes en spécialisant les arguments

```
public boolean equals(Point pt) {
  return this.x == pt.x && this.y == pt.y;
}
```

surcharge (overloads) la méthode `equals(Object o)` héritée de `Object`

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2);    --> true
Object o = p2;
p1.equals(o)      --> false
o.equals(p1)      --> false
```

Le choix de la méthode à exécuter est effectué
statiquement à la compilation
en fonction du type déclaré de l'objet récepteur du message et du type déclaré du (des) paramètre(s)

instanceof

- Format : objectReference instanceof type, renvoie un boolean

```
String s = "Miage" ;
if(s instanceof String) { ...}
Object o = new String("egaiM") ;
if(o instanceof String) {...}
if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
String s = "Miage" ;
if(s instanceof Object) { ...} //vrai
```

```
String s = "Miage" ;
if(s instanceof Integer) { ...} //compile pas
```

```
String s = null ;
if(s instanceof String) { ...} //faux
```

A PROPOS DE EQUALS

EXEMPLE TIRÉ DU COURS DU PROFESSEUR PHILIPPE GENOUD

- Tester l'égalité de deux objets de la même classe

```
public class Object {
 ...
 public boolean equals(Object o)
   return this == o
 }
 ...
}
```

```
public class Point {

      private double x;
      private double y;

      ...

}
```

```
@Override
public boolean equals(Object o) {
    if (this == o)
         return true;
    if (! (o instanceof Point))
         return false;
    Point pt = (Point) o; // downcasting
    return this.x == pt.x && this.y ==  pt.y;
}
```

redéfinir (overrides) la méthode equals(Object o) héritée de Object

```
Point p1 = new Point(15,11);
Point p2 = new Point(15,11);
p1.equals(p2)      --> true
Object o = p2;
p1.equals(o)       --> true
o.equals(p1)       --> true
```

EXERCICES

Etant donné que la classe Triangle étend la classe Figure, Dites pour chacune de ces lignes si elle est correcte, sinon pq ?

(a)
```
Triangle x= new Triangle() ;
Object y = (Object)x ;
Triangle z=y ;
```

(b)
```
Figure y =new Figure() ;
Triangle x= (Triangle)y ;
Figure z=x ;
```

(c)
```
Triangle x= new Triangle() ;
Figure y = x ;
Triangle z=(Triangle)y
```

(d)
```
Figure y =new Figure() ;
Triangle x= (Triangle)y ;
Figure z=(Figure)x ;
```

EXERCICES

Etant donné que la classe Triangle étend la classe Figure, Dites pour chacune de ces lignes si elle est correcte, sinon pq ?

(a)
```
Triangle x= new Triangle() ;
Object y = (Object)x ;
Triangle z=y ; //erreur au niveau de la compilation y de type Object pas classe fille de Triangle
```

(b)
```
Figure y =new Figure() ;
Triangle x= (Triangle)y ; //tout se passe bien à la compilation mais erreur à cette instruction au niveau de l'exécution une levée d'un exception ClassCastException car la promesse n'est pas tenue: une figure n'est pas classe fille de triangle
Figure z=x ;
```

(c)
```
Triangle x= new Triangle() ;
Figure y = x ;
Triangle z=(Triangle)y ;//Ici tout va bien : l'objet construit par la première instruction est un Triangle, la deuxième instruction est un upcasting implicite, la troisième instruction est un down-casting explicite qui se déroule bien.
```

(d)
```
Figure y =new Figure() ;
Triangle x= (Triangle)y ;
Figure z=(Figure)x ;
```