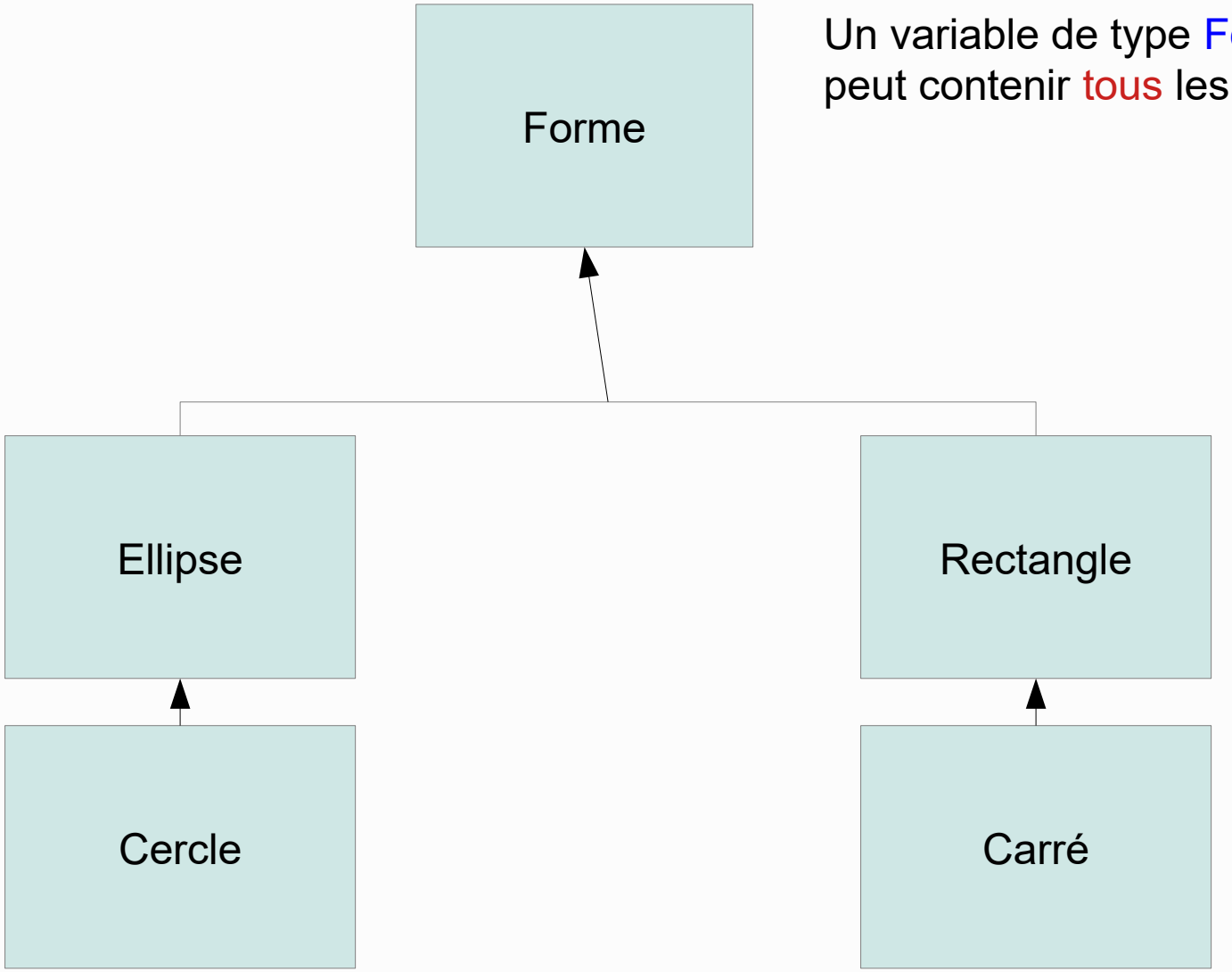
Le polymorphisme

- Concept important et complexe en POO
- Le polymorphisme indique qu'une variable peut contenir des objets de type différent
- En Java, le polymorphisme est contrôlé par l'héritage
- Durant l'exécution, une variable de type classe C peut :
  - contenir null
  - contenir un objet de type C
  - un objet d'une classe dont C hérite
  - Ce typage dynamique peut évoluer durant l'exécution
  - Par exemple, une variable de type Rectangle peut contenir un objet de type Carré
  - Si r est de type Rectangle et c de type Carré
  - r ← c est licite mais pas c ← r

Exemple



Un variable de type Forme peut contenir tous les objets

Conversion implicite

- La conversion vers une classe ancêtre est implicite :
  - affectation à une variable ou attribut
  - passage en tant qu'argument à une méthode
  - appel à une méthode héritée

```
public class Test {
    public static void main(String [] args){
    Carre c = new Carre(5.0);
    Rectangle r = new Rectangle(4.0,3.0);
    Rectangle r2;
    r2 = c; //ok
    Carre c2;
    c2 = r; // erreur
    }
}
```

r2 est de type Rectangle
mais a comme typage dynamique Carre

Un terme savant (polymorphisme)

- Carré hérite de Rectangle, alors un objet de type Carré peut être considéré comme un Rectangle
- On peut également écrire
  - Rectangle r = new Carré() ;
- Le polymorphisme permet de considérer un objet d'une classe dérivée comme un objet d'une classe base
- De spécifique vers moins spécifique (vers le haut dans la hiérarchie des classes) est sans risque
  - il ne s'agit pas réellement d'une conversion : l'objet n'est pas modifié (c'est toujours un Carré même s'il est stocké dans un objet de type Rectangle)

Conversion explicite de type

- Java s'assure toujours qu'une variable de type C contient toujours un objet de type C ou un objet dérivé de la classe C
- Inconvénient : on ne peut accéder qu'aux méthodes et attributs de la classe C
- Une solution le transtypage explicite (Classe) objet
- Si l'objet n'est pas du type Classe ou un ancêtre de Classe
  - erreur d'exécution → exception

Exemple

- Rectangle r = new Carré() ;
- Carré c = (Carré) r ;

- On considère que r est en fait un objet Carré même s'il est dans une variable de type Rectangle
- Il faut dans ce cas un transtypage explicite
- Il faut vérifier à l'exécution que cette conversion est possible
- Cette conversion n'est jamais implicite (on ne transforme jamais un Rectangle en Carré sans prendre des précautions)

Transtypage explicite

- Premier essai
  Etudiant et2 = new Personne() ;
- Nous obtenons une erreur quand nous compilons le code. Toute Personne n'est pas un Etudiant (ça ne fonctionne pas dans ce sens)
- Deuxième essai
  Etudiant et2 = (Etudiant) new Personne() ;
- Ça compile. Attention, nous aurons une exception à l'exécution. La raison est que la partie droite n'est pas un Étudiant (si c'en était un, cela aurait marché...)

Liaison dynamique

- En java, lors d'un appel de méthode, c'est toujours le type dynamique qui est utilisé
- Pas toujours facile de déterminer quel code est exécuté
- Le typage statique (écrit dans le code source) ne sert qu'à vérifier que la méthode existe
- Exemple : Que donne le code suivant ?

```
public class TestLiaison {

   public static void main(String [] args){

      Carre c1 = new Carre(5);
      System.out.println(c1);

      Rectangle r1 = new Rectangle(5,5);
      System.out.println(r1);

      Rectangle r2 = new Carre(10);
      System.out.println(r2);
}}
```

Liaison Dynamique

- Autre exemple

```
ArrayList<Forme>  t = new ArrayList<Forme>() ;
t.add(new Rectangle(3,10)) ;
t.add( new Carré(7) );
t.add(new Carré(5)) ;
t.add(new Rectangle(8,15)) ;
for(int i = 0 ; i < t.size() ; i++)
 System.out.println(t.get(i)) ;
```

On définit une méthode toString() dans les classe Carré, Rectangle et Forme
C'est le typage dynamique qui décide de la méthode à appeler

- A l'exécution :

```
je suis un rectangle de taille 10.0x3.0
je suis un carré de côté 7.0
je suis un carré de côté 5.0
je suis un rectangle de taille 15.0x8.0
```

Principe de l'héritage

- Si l'objet est de type dynamique C, on regarde dans les classes ancêtres de C si la méthode existe
- Si on enlève les méthodes toString() de Carré et Rectangle

→ Forme est ancêtre

```
je suis une forme géométrique
je suis une forme géométrique
je suis une forme géométrique
je suis une forme géométrique
```

La méthode appelée est celle de la classe mère Forme

Attention : typage statique

- Si on utilise la méthode presentation() définie dans la classe Rectangle et la classe Carre mais mais pas dans la classe Forme

```
public static void main(String [] args){
  ArrayList<Forme>  t = new ArrayList<Forme>();
  t.add(new Rectangle(3,10));
  t.add( new Carre(7));
  t.add(new Carre(5));
  t.add(new Rectangle(8,15));
  for(int i = 0; i < t.size(); i++)
    t.get(i).presentation();
  }
```

```
error: cannot find symbol
          t.get(i).presentation();
                   ^
 symbol:   method presentation()
 location: class Forme
```

Une méthode n'est compilable que si elle est définie dans la classe
Ici presentation() n'est pas définie dans forme (typage statique)
Ici, la liaison statique est vérifiée à la compilation en utilisant
le type statique (Forme)

en Java on peut utiliser

```
Rectangle monR ;
Carré monC ;
...
if (monR instanceof Carré)
   monC = (Carré) monR ;
```

Note sur les attributs dans l'héritage

- On redéfinit et on surcharge très régulièrement les méthodes lors de l'héritage
- On peut redéfinir les attributs dans la classe héritée mais c'est en général très rare
- L'attribut redéfini masque bien évidemment l'attribut initial
  - si les attributs sont définis comme privés (bonne programmation), il n'y a pas de confusion
  - cependant la résolution de l'attribut est fait **statiquement** et pas dynamiquement
  - conclusion :
    - on préfère les attributs privés
    - si on utilise des attributs protégés on ne les redéfinit pas

Exemple

```java
public class Attribut{

    public int valeur;

    public int getValeur(){
        return valeur;
    }
}
```

```java
public class AttributHerite extends Attribut {

    public int valeur;

    public int getValeur(){

        return valeur;
    }

    public static void main(String [] args){

        Attribut h1 = new Attribut();
        h1.valeur = 2;

        System.out.println(h1.getValeur());

        Attribut h2 = new AttributHerite();
        h2.valeur = 2;
        System.out.println(h2.getValeur());
    }
}
```

Que donne l'exécution ?

Héritage et méthode final

- Dans le cas de l'héritage :
  - une méthode peut-être appelée par une sous-classe
  - une méthode peut être redéfinie dans une sous-classe

Exemple classique :

- on arrive à obtenir le mdp !

Solution :

le mot-clef final interdit la redéfinition d'une méthode dans une sous-classe

```
public class MaBanque {
    public  String entrerMdP(){
        // saisie avec des  *
    }
}
```

```
public class TresMalin extends MaBanque {
    public  String entrerMdP(){
        String s;
        s = super.entrerMdP();
        System.out.println(s);
        return s;
    }
}
```

Note : une classe peut être final System, String, ...

Héritage - Polymorphisme

- Héritage : capitalisation des membres communs dans une classe mère
  - Classe générique (mère, super-classe) : propriétés et comportements communs
  - classe spécifique (enfant) : propriétés et comportements spécifiques
  - les classes 'enfant' héritent des propriétés et comportements communs visibles
- Polymorphisme de redéfinition : possibilité de redéfinir des méthodes dans
  - les classes 'enfant'
  - Les méthodes sont redéfinies dans les classes 'enfant' afin de redéfinir un comportement spécifique, différent du comportement de la classe mère
  - Les méthodes redéfinies ont la même signature

Les classes abstraites

- Nous souhaitons faire dériver toutes nos classes représentant des formes géométriques d'une unique classe Forme pour
  - factoriser les attributs communs (couleur, nombre de côtés, nom...)
  - factoriser le code des méthodes indépendantes du type de la forme (déplacer, colorier, …)
  - créer des tableaux dynamiques de Forme
  - donner des signatures des méthodes qui seront implémentées dans les classes dérivées (surface, périmètre, ...)
  -

Première solution

- Certains méthodes ne peuvent être écrites que dans les classes dérivées
  - surface d'un cercle disponible uniquement dans la classe Cercle
  - surface d'un rectangle uniquement dans la classe Rectangle
- On ne peut avoir un tableau dynamique de Forme et vouloir calculer la surface (liaison statique → nécessité de la méthode surface())
- On écrit une une pseudo-méthode surface qui sera redéfinie dans les classes dérivées

Solution inélégante

```
public  class Forme {

    private String nom;
    private String couleur;

    public Forme(String n){
        nom = n;
    }

    public double surface(){
        System.out.println("Méthode implémentée dans les sous-classes ");
        return 0.0;
    }


    public String toString(){
        return "Je suis une forme géométrique";
    }

    //public void presentation(){
    //  System.out.println("je suis une forme géométrique");
    //}

}
```

La solution : les classes abstraites

- Les classes abstraites (mot-clef abstract) répondent à ce besoin :
  - elles ne sont pas instanciables (new Forme()) impossible
  - des méthodes peuvent être écrites dans la classe abstraite
  - les méthodes déclarées abstraites (abstract) :
    - ne sont **pas** implémentées dans la classe abstraites
    - sont **obligatoirement** implémentées dans les classes dérivées
  - Il est possible de définir un constructeur pour une classe abstraite

Illustration

```
public  abstract class FormeAbstraite{

    private String nom;
    private String couleur;

    public FormeAbstraite (String n){
        nom = n;
    }

    public abstract double surface();

}
```

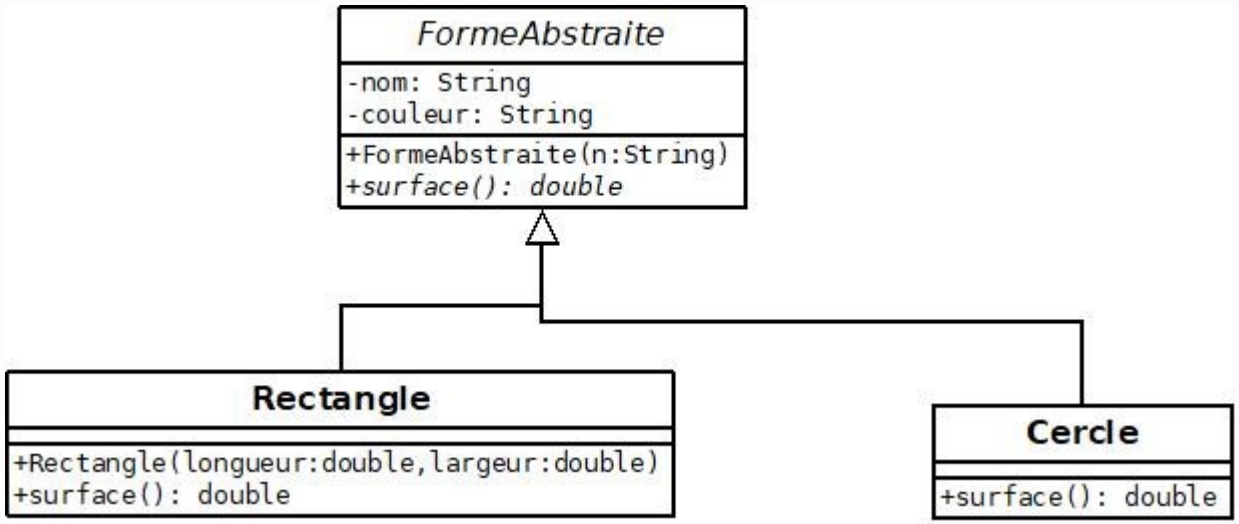
Note :

```
public static void main(String [] args){

        FormeAbstraite f = new FormeAbstraite("cercle");

    }
```

```
FormeAbstraite.java:20: error: FormeAbstraite is abstract; cannot be instantiated
    FormeAbstraite f = new FormeAbstraite("cercle");
```

En UML

- Les classes abstraites et les méthodes abstraites sont en italiques
- Toute classe possédant une méthode abstraite est abstraite



Synthèse

- Une classe est instantiable
  - possède un ou plusieurs constructeurs
  - des attributs
  - des méthodes avec du code
  - peut être dérivée
- Une classe abstraite est incomplète (complétée par dérivation)
  - peut avoir un constructeur (ou plusieurs)
  - possède des attributs
  - des méthodes avec code
  - des méthodes sans code
  - DOIT être dérivée

Bien présenter ses programmes

- Découverte de l'outil javadoc
- Après javac (compilateur), java (appel de la JVM), c'est le 3ème outil principal de l'environnement java
- Il permet de générer automatiquement de la documentation sur les classes en respectant un certain formalisme

Les interfaces

- La classe est un moule qui décrit les objets créées, les méthodes permettant de les manipuler
- L'interface est une abstraction qui décrit la liste des opérations que l'on peut faire sur un objet

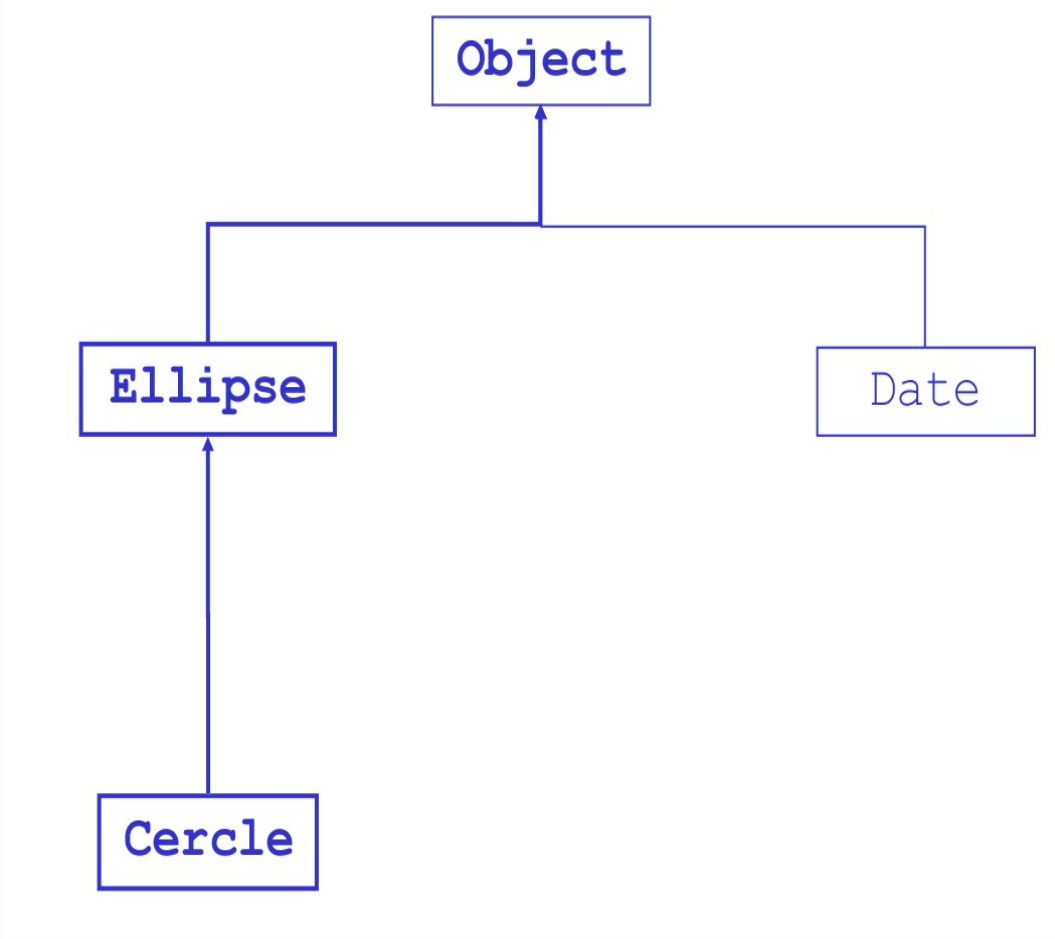
Java possède la notion d'interface

Pour une interface :

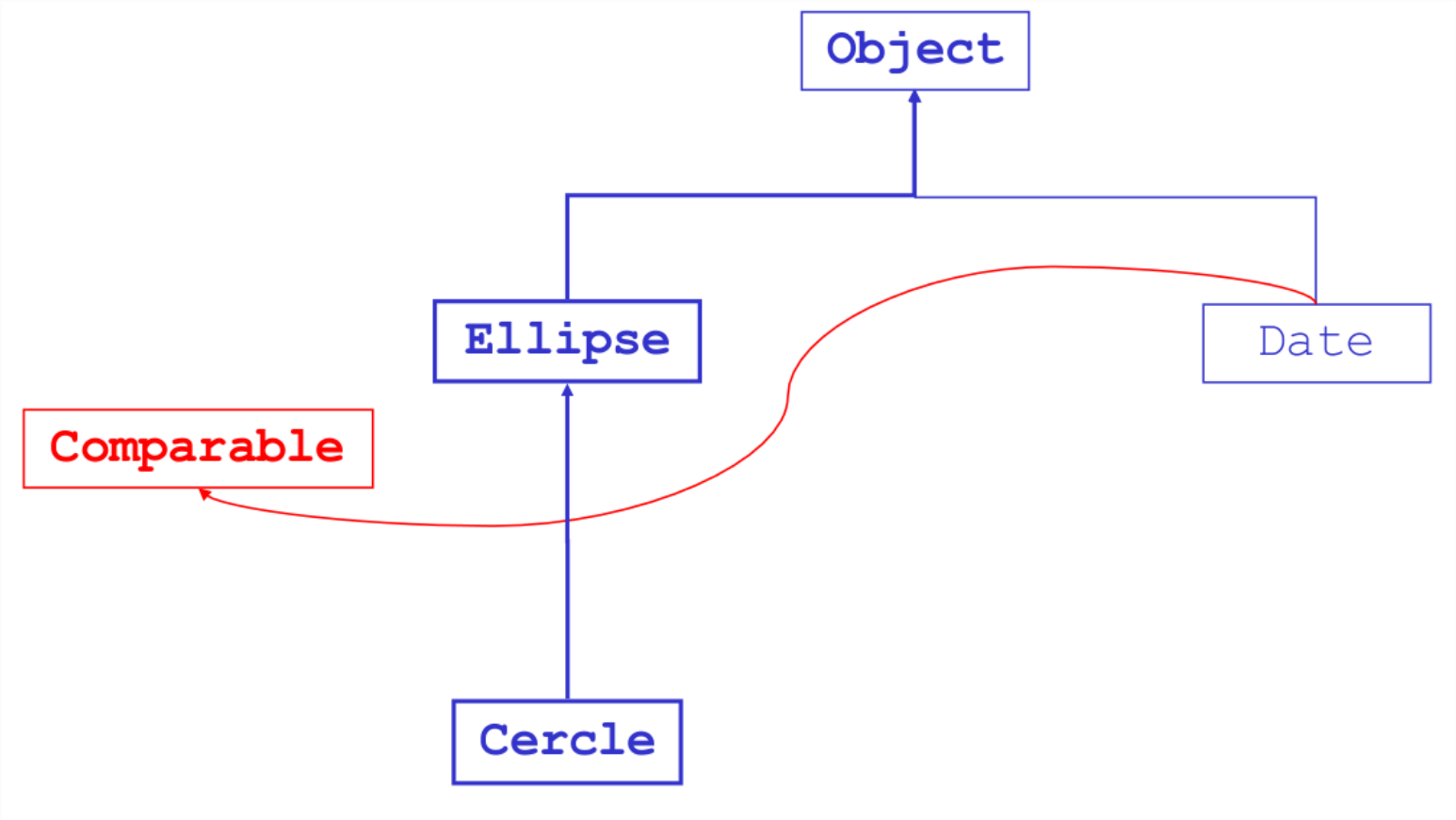
  - Pas d'attributs sauf les attributs static final (pour les constantes)
  - Pas de constructeurs (on ne construit pas d'instances)
  - Méthodes publiques mais uniquement leur signature (pas de code)

- Les classes peuvent décider d'implémenter des interfaces (on parle de contrat)
- Les interfaces sont des propriétés transverses des classes

Les interfaces



Les dates sont comparable(s)



compact1, compact2, compact3
java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedDateTime<D>, Delayed, Name, Path, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, AccessMode, AclEntryFlag, AclEntryPermission, AclEntryType, AddressingFeature.Respon
Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, CertPathValidatorException.BasicReason, Character,
Character.UnicodeScript, CharBuffer, Charset, ChronoField, ChronoUnit, ClientInfoStatus, CollationKey, Collector.Characteristics,
Component.BaselineResizeBehavior, CompositeName, CompoundName, CRLReason, CryptoPrimitive, Date, Date, DayOfWeek, Desktop.Action, Diagnostic.Kind,
Dialog.ModalExclusionType, Dialog.ModalityType, DocumentationTool.Location, Double, DoubleBuffer, DropMode, Duration, ElementKind, ElementType, Enum,
FileTime, FileVisitOption, FileVisitResult, Float, FloatBuffer, FormatStyle, Formatter.BigDecimalLayoutForm, FormSubmitEvent.MethodType,
GraphicsDevice.WindowTranslucency, GregorianCalendar, GroupLayout.Alignment, HijrahChronology, HijrahDate, HijrahEra, Instant, IntBuffer, Integer, Iso
IsoEra, JapaneseChronology, JapaneseDate, JavaFileObject.Kind, JDBCType, JTable.PrintMode, KeyRep.Type, LayoutStyle.ComponentPlacement, LdapName, Link
LocalDate, LocalDateTime, Locale.Category, Locale.FilteringMode, LocalTime, Long, LongBuffer, MappedByteBuffer, MemoryType, MessageContext.Scope,
MinguoChronology, MinguoDate, MinguoEra, Modifier, Month, MonthDay, MultipleGradientPaint.ColorSpaceType, MultipleGradientPaint.CycleMethod, NestingK
Normalizer.Form, NumericShaper.Range, ObjectName, ObjectStreamField, OffsetDateTime, OffsetTime, PKIXReason, PKIXRevocationChecker.Option, PosixFileP
ProcessBuilder.Redirect.Type, Proxy.Type, PseudoColumnUsage, Rdn, ResolverStyle, Resource.AuthenticationType, RetentionPolicy, RoundingMode,
RowFilter.ComparisonType, RowIdLifetime, RowSorterEvent.Type, Service.Mode, Short, ShortBuffer, SignStyle, SOAPBinding.ParameterStyle, SOAPBinding.Sty
SOAPBinding.Use, SortOrder, SourceVersion, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, StandardCopyOption, StandardLocation, StandardOper
StandardProtocolFamily, String, SwingWorker.StateValue, TextStyle, ThaiBuddhistChronology, ThaiBuddhistDate, ThaiBuddhistEra, Thread.State, Time, Time
TimeUnit, TrayIcon.MessageType, TypeKind, URI, UUID, WebParam.Mode, Window.Type, XmlAccessOrder, XmlAccessType, XmlNsForm, Year, YearMonth, ZonedDate
ZoneOffset, ZoneOffsetTransition, ZoneOffsetTransitionRule.TimeDefinition

---

public interface **Comparable<T>**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo met
referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be us
in a sorted map or as elements in a sorted set, without the need to specify a comparator.

public int compareTo(Object o)

- compare l'objet this avec l'objet spécifié afin de l'ordonner
- retourne un entier négatif, zéro ou un entier positif si l'entier est plus petit, égale ou plus que l'objet o

- Paramètres

l'objet o à comparer

- Retourne

un entier négatif, ou en entier positif ou zéro

Method Detail

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specifie

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

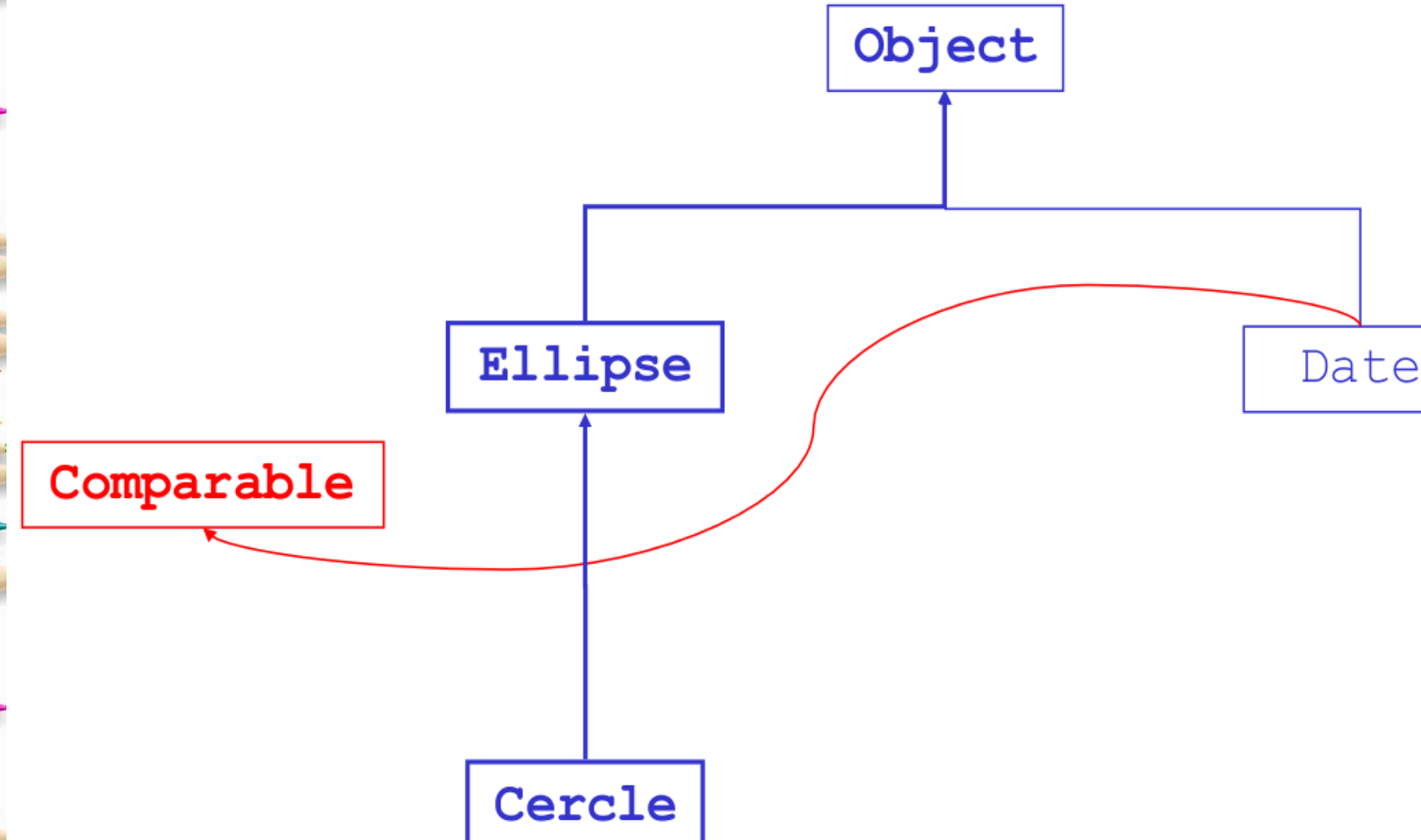
`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

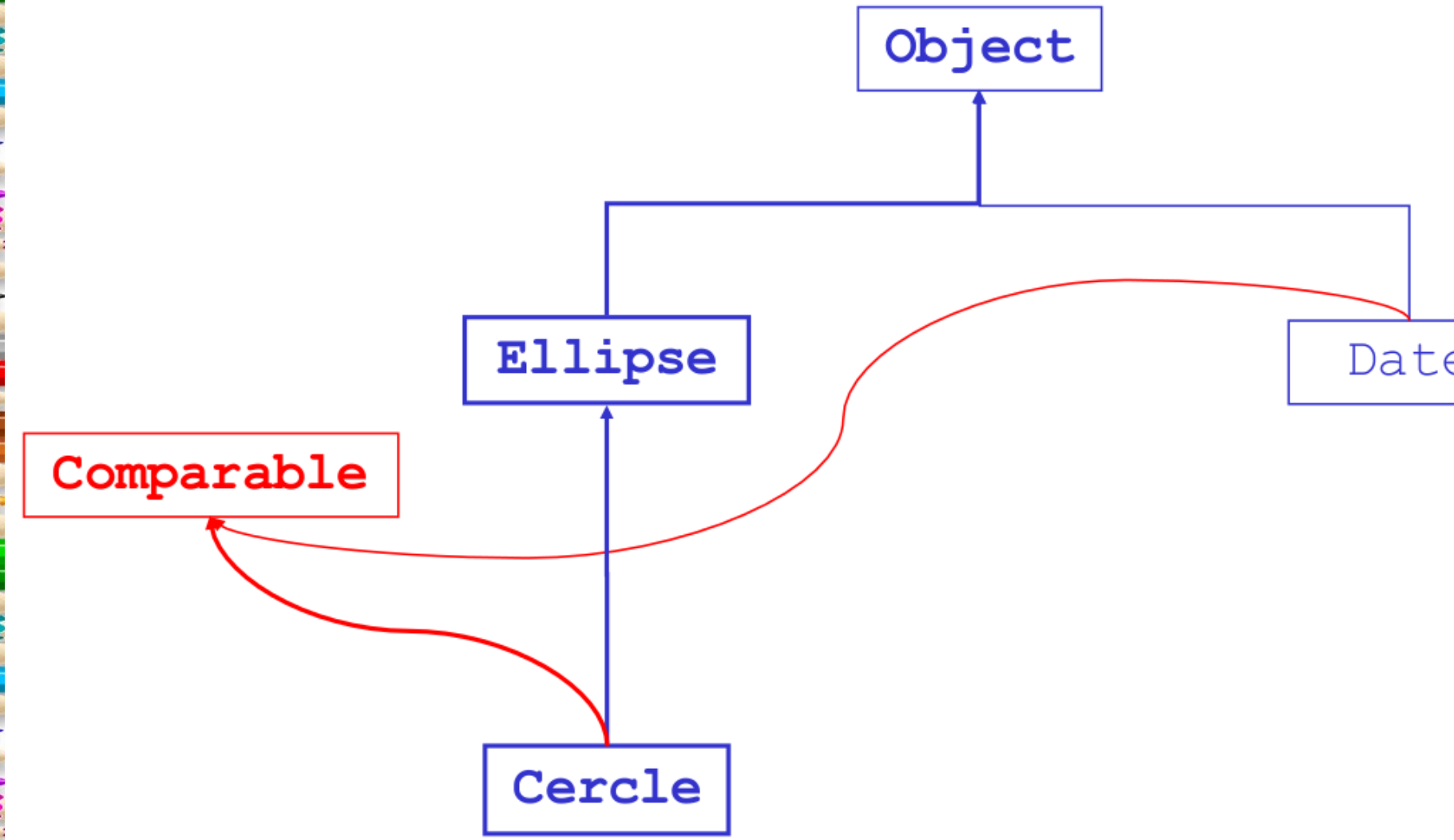
Throws:

`NullPointerException - if the specified object is null`

`ClassCastException - if the specified object's type prevents it from being compared to this object.`

Les Cercles deviennent Comparable





Les Cercles deviennent Comparable

```
public class Cercle implements Comparable {


    public int compareTo(Object o){

        double monR = this.getRayon();
        double autreR = ((Cercle) o).getRayon();

        if (monR < autreR)
            return -1;

        if (monR == autreR)
            return 0;
        else
            return +1;
    }

}
```

Interface des objets « Comparable »

- public interface Comparable
- Une seule méthode
  public int compareTo(Object o) ;
-
- Implémentée par les classes :
  Character, String, Integer, Float, Double, Date,...

Exemple : les listes

- La bibliothèque standard Java contient des structures de données très utile comme les listes
- c'est en fait une interface : java.util.List
- Cette interface est implémentée dans les ArrayList, les LinkedList,… (en plus ce sont des interfaces génériques E!)
- Les opérations sont identiques sur toutes ces implémentations
  - boolean add(E) → ajouter un élément
  - E get(int) → accès
  - void clear() → raz

Les listes suites

- Il est possible et conseillé de déclarer une variable de type interface au lieu du type final
- → cela limite la dépendance à l'implémentation

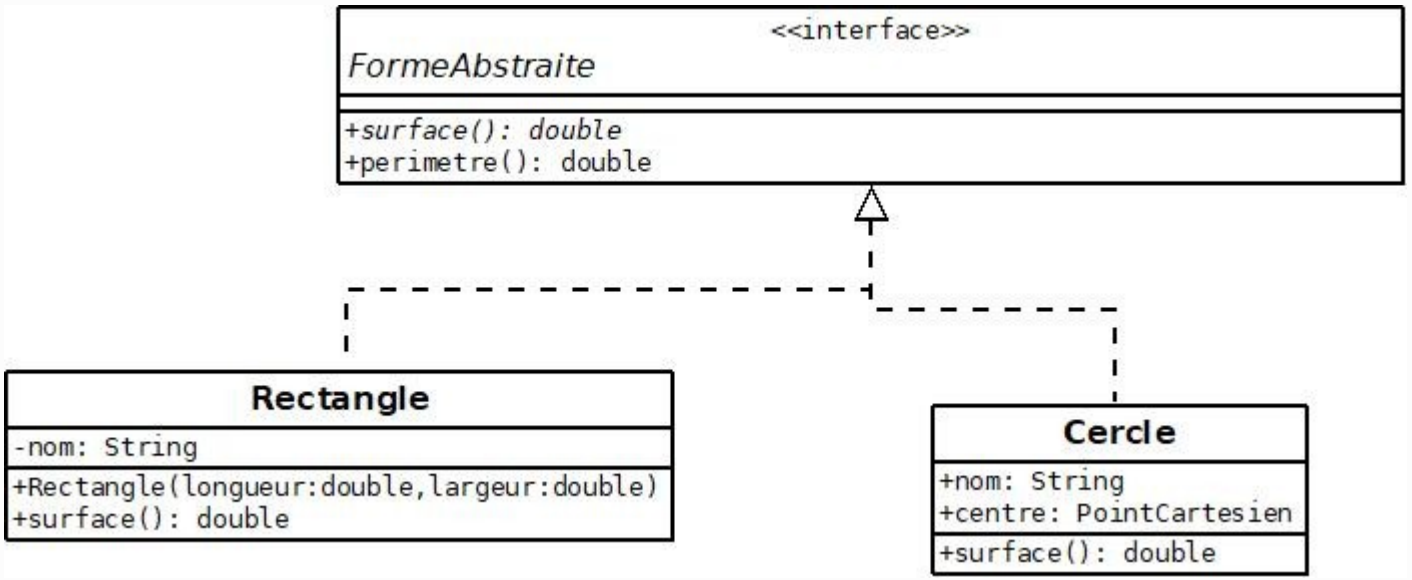
```
List<PointCartesien> listePoints = new
ArrayList<PointCartesien>() ;
```

plutôt que

```
ArrayList<PointCartesein> listePoints = new
ArrayList>PointCartesien>() ;
```

Diagramme UML d'une interface

- Une interface est matérialisée par le mot-clef <<interface>>
- L'implémentation dans une classe est une flèche creuse en pointillés



Notes sur les interfaces

- A la différence d'une classe qui ne peut hériter que d'une classe mère
- Il est possible d'implémenter plusieurs interfaces (sorte d'héritages multiples) dans une classe
- Java utilise énormément le concept d'interfaces
- Règle : il faut utiliser une interface plutôt que son implémentation
- List<PointCartesien> plutôt que ArrayList<PointCartesien>
- on fait le moins d'hypothèses possibles sur la représentation des objets
- Pas toujours facile

La généricité

Les types génériques

Qu'est-ce qu'un type générique ?

- Un type générique est une classe (mais peut être également une interface) qui prend un type (de type objet)
- L'exemple suivant est une classe mémorisant un obhet de type Integer
-

```
public class MémoireInteger {

    private Integer mem;

    public void set(Integer i){
        mem = i;
    }

    public Integer get(){
        return mem;
    }
}
```

Question ?

- Que faut-il faire pour stocker des objets d'autres types :
  - Float, Rectangle, Carré,… ?
- Deux possibilités :
  - Écrire une classe MémoireFloat, MémoireRectangle, …
  - Écrire une **classe générique**

```
public class Mémoire<T> {

    private T mem;

    public void set(T o){
        mem = o;
    }

    public T get(){
        return mem;
    }
}
```

Syntaxe - Utilisation

- La syntaxe public class Mémoire<T> permet de passer en paramètre un type objet à la classe
- Pour utiliser la classe, il suffit de l'instancier avec le type entre < >

```
public class TestMemoire {

   public static void main(String [] args){

      Mémoire<Integer> memInt = new Mémoire<Integer>();
      Mémoire<Double> memDouble = new Mémoire<Double>();
      Mémoire<Rectangle> memRectangle = new Mémoire<Rectangle>();

      memInt.set(5);
      memDouble.set(2.37);
      memRectangle.set(new Rectangle(5,7));

      System.out.println(memRectangle.get());

   }


}
```

Remarques

- La syntaxe < > permet de créer le type lors de la compilation et de déterminer le type à l'exécution
- Il est possible créer des classes avec plusieurs types génériques
- Par convention, on n'utilise qu'une seule lettre majuscule pour désigner un type d'une classe générique
- Cette syntaxe doit vous rappeler des exemples déjà vus en cours comme la définition d'une ArrayList
- Les classes Collections en java utilisent la généricité

Question

- Si on regarde la classe List et ArrayList
- On voit que ArrayList est une sous-classe de List
- Vaut-il mieux écrire ?
- ArrayList<Integer> listeInt = new ArrayList<Integer>() ;
- Ou
- List<Integer> listeInt = new ArrayList<Integer>() ;
-
- Les deux écritures sont correctes