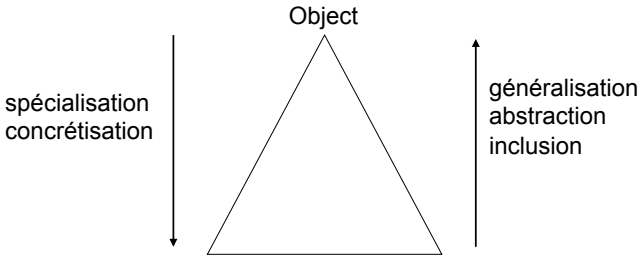
Abstraction

Classes et méthodes abstraites
Polymorphisme
Sous-typage

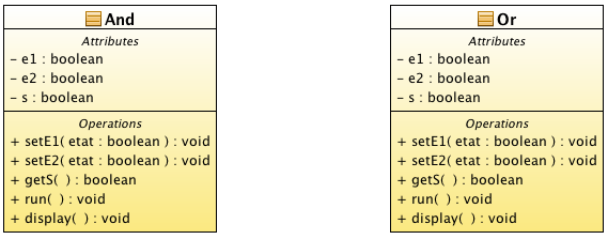
Hiérarchie (arbre) de classes



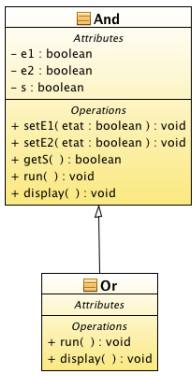
	classe	sous-classe
interprétation extensionnelle	ensemble (type)	inclusion (sous-type)
interprétation intensionnelle	schéma de code	généralisation/ spécialisation abstraction/ concrétisation

Abstraction

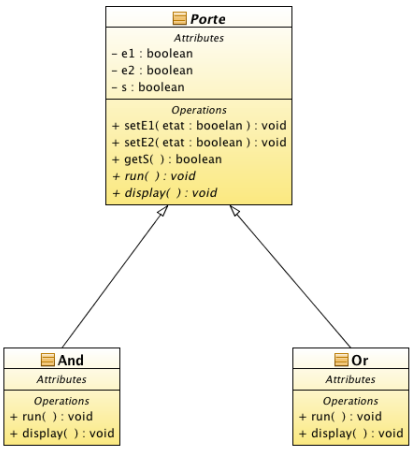
- A partir de plusieurs classes semblables, abstraire une sur-classe commune
- Factorisation de code



Erreur de conception…



Solution: surclasse abstraite



Classe abstraite

```
abstract class Porte {

// variables d'instance
  boolean e1, e2, s;

// methodes
      void setE1(boolean etat) {e1 = etat;}
      void setE2(boolean etat) {e2 = etat;}
      boolean getS() {
            run();
            return s;}
      abstract void run();
      abstract void display();
}
```

Classe abstraite

```
class And extends Porte {
      void run() {
            s = e1 && e2;
      }
      void display() {...}
}

class Or extends Porte {
      void run() {
            s = e1 || e2;
      }
      void display() {...}
}
```


Abstraction

- **Classes et méthodes abstraites**
  - Classe abstraite = non instanciable
  - Spécifie des méthodes abstraites, implantées dans les sous-classes
- **Méthodes génériques**

  Dans la sur-classe, les méthodes qui font référence par this-message à des méthodes abstraites sont implicitement «génériques » pour les sous-classes.

Méthodes génériques

```
abstract class Porte {
  boolean getS() { // generique
      this.run();
      return s;
  }
  abstract void run();
...}
{// programme utilisateur
  And a = new And();
  Or o = new Or();
  a.setE1(true);
     a.getS() // --> false
  o.setE1(true);
     o.getS() // --> true...}
```


Polymorphisme, hiérarchie de classes et typage

- Polymorphisme

  une même opération peut être définie différemment dans des classes distinctes.

- Polymorphisme de surcharge
  - Classes incomparables
  - exemples

```
a.display(); // And:display()
rec.display(); // Rectangle:display()
```

- Polymorphisme de redéfinition (ou d'inclusion)

```
a.getS(); // Porte:getS() -> And:run()
o.getS(); // Porte:getS() -> Or:run()
```

Polymorphisme, hiérarchie de classes et typage

- **Hiérarchie de classes => hiérarchie de types**
  - Tout objet instance d'une classe peut être considéré du type de ses sur-classes
  - Ou inversement : partout où l'on attend un objet d'une classe donnée, tout objet d'une sous-classe convient
- **Variable polymorphe**

  Soit x une variable de type C, x peut référencer:
  - tout objet instance de C (typage "fort" classique)
  - mais aussi tout objet instance d'une sous-classe de C (**typage souple**)

Affectation polymorphe : Exemple

```
Porte p;
And a1 = new And(), a2;
Or o;
Rectangle r;

// affectations valides : typage fort classique
a2 = a1 ;
// affectations non valides : '' horizontales ''
a2 = o;
p = r;

// typage souple
// affectations '' verticales '' toujours valides : upcast
p = a1; p = o;
// affectations ''verticales '' hypothétiques : downcast
a2 = p; // non valide en général sauf...
a2 = (And)p; // downcast valide si...
if (p instanceof And) a2=(And)p; // sinon ClassCastException
```

Variable et liaison dynamique

- **Type statique** d'une variable
  = type de la **déclaration**
  il détermine, à la **compilation**, les opérations applicables (dont les abstract déclarées)
- **Type dynamique** d'une variable
  = type de la **valeur** à l'**exécution**
  = type de l'objet référencé
  - il détermine les opérations effectivement appliquées (parmi celles applicables)
  - **liaison dynamique** des méthodes
- Ceci s'applique à toute catégorie de variable (`this`, variables d'instance, locales, paramètres, indexées (tableaux), ...)

Liaison dynamique sur this

```
abstract class Porte {
  boolean getS() { // generique
      this.run();
      // type statique de this = Porte
      return s;
  }
  abstract void run();
...}
{// programme utilisateur
  And a = new And();
  Or o = new Or();
  a.setE1(true);
     a.getS() // => this.run()
      // type dynamique de this = And
  o.setE1(true);
     o.getS() //=> this.run()
      // type dynamique de this = Or
```

Liaison dynamique sur paramètres

```
// exemple de procedure dans une application
// utilisatrice de Porte's ...
boolean test(Porte p) {
  p.setE1(true);
  return p.getS();
}

And a = new And();
Or o = new Or();
Porte p;
test(a); // --> false
test(o); // --> true
p = quellePorte();
test(p);
```


Liaison dynamique : SD

- Une structure de données (tableau, liste, ...) peut contenir des objets de toute sous-classe (type dynamique) de la classe déclarée (type statique) pour ses éléments.
- SD hétérognènes

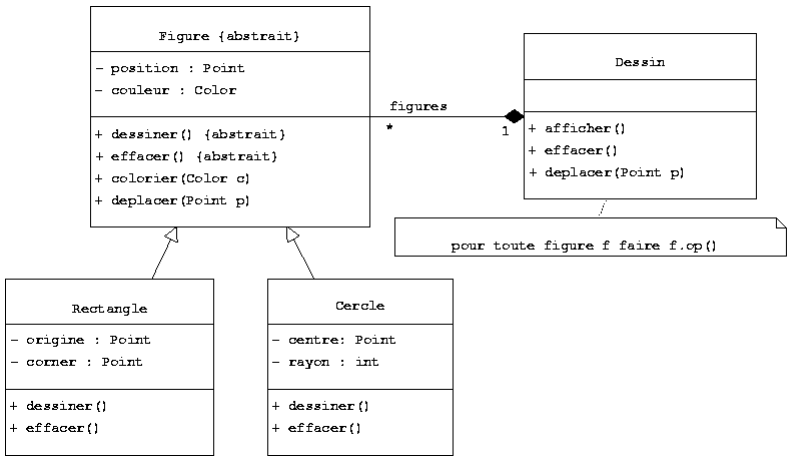
```
Exemple
Porte[] circuit = new Porte[n]; // type statique

circuit[0] = new And(); // types dynamiques ...
circuit[1] = new Or();
circuit[2] = new Nand();...

for (int i=0;i<n;i++) {
  circuit[i].setE1(false);
  circuit[i].setE2(false);
  circuit[i].run();
}
```

```
for (Porte p : circuit) {
  p.setE1(false);
  p.setE2(false);
  p.run(); //polymorphe
}
```

Etude de cas

- *Décrire des figures (rectangles, cercles, ...)*
  - *colorées, et dont on doit pouvoir changer la couleur*
  - *positionnées, et que l'on doit pouvoir effacer et déplacer.*
- Identification des objets
  - Rectangle
  - Cercle, …
- Munis du même protocole :
  - dessiner()
  - effacer()
  - colorier(Color c)
  - deplacer(Point p)
- Mêmes spécifications
  => sur-type commun : Figure

Etude de cas (suite)



Etude de cas (suite)

```
abstract class Figure {
// champs
  Point position;
  Color couleur;
// methodes
      abstract void dessiner ();
      abstract void effacer ();
      void colorier(Color c) { // generique
            couleur=c;
            this.dessiner();}
      void deplacer(Point p) { // generique
            this.effacer();
            position.translater(p);
            this.dessiner();}
}
```


Etude de cas (suite)

```
class Rectangle extends Figure {
  Point origine, corner;
  void dessiner() {...}
  void effacer() {...}
}

class Cercle extends Figure {
  Point centre;
  int rayon;
  void dessiner() {...}
  void effacer() {...}
}

...
```

Etude de cas (suite)

- *Un dessin est formé de figures. On doit pouvoir afficher, effacer et déplacer un dessin.*
- Identification

  Dessin :
  - afficher()
  - effacer()
  - deplacer(Point p)
- Structuration
  - Un dessin => une liste de Figure.
  - Algorithmes génériques sur les opérations afficher, effacer, deplacer :

  `pour toute Figure f faire f.operation() fait`

Etude de cas (suite)

```
class Dessin {
// structure de donnees a voir...
  Figure figures[] = new Figure[N];
  int nbFigures;
// methodes
  void afficher() {
      for(int i=0;i<nbFigures; i++)
            figures[i].dessiner(); // polymorphe
  }
  void deplacer(Point p) {
      for(int i=0;i<nbFigures; i++)
            figures[i].deplacer(p); // polymorphe
  }
}
```

Qualités logicielles

- Extensibilité
  - Ajout d'un nouveau type de figure (Triangle)
  - Incrémental et modulaire (sans retouche du code existant)
- Réutilisation
  - Le code de Figure est réutilisable dans le nouveau sous-type
  - Programmation synthétique
- Généricité
  - Les portions de codes (applications) écrites à un niveau de la hiérarchie de classes sont applicables à toutes les sous-classes
  - les programmes restent applicables à toute nouvelle sous-classe

Si l'héritage n'existait pas…

- 1ère solution
  - Ensemble de types à plat et définir des opérations différentes: {Rectangle, Cercle, …} X {dessiner, effacer, …}
  - Pas de sur-type Figure => on ne peut regrouper les entités de types différents dans une même SD
- 2ème solution
  - structures à champs variants
  - record Pascal ou ADA, unions C
  - type et programmation «tagués»
- Qualités
  - Permet de simuler « à la main » le polymorphisme et la généricité
  - Peu efficace et risque d'erreur
  - Peu modulaire, maintenable et extensible

Programmation « taguée »

```
type Tags = (rectangle, cercle);

type figure (tag : Tags) = structure
  position : Point;
  couleur : Color;
  cas tag =
     cercle : rayon : int; centre : Point;
     rectangle : origine, corner : Point;

type Figures = tableau[n] de Figure;
```


Programmation « taguée »

```
// polymorphisme « à la main »

procedure dessiner (x : Figure)
  cas x.tag =
     rectangle : ... code ....
     cercle : ... code …

procedure effacer (x : Figure)
  cas x.tag =
     rectangle : ... code ....
     cercle : ... code ...
```

Programmation « taguée »

```
// généricité de Figure « simulée »
procedure deplacerFigure(f : Figure, p : Point)
  effacer(f);
  translater(f, p);
  dessiner(f)
procedure colorierFigure ...

// généricité de l'application Dessin « simulée »
procedure afficherDessin (f: Figures)
      pour i de 1 a n faire dessiner(f);
procedure deplacerDessin (f: Figures, p: Point)
      ...
```