

# Programmation orientée-objet

Langage Java

Stéphane Lopes

[stephane.lopes@uvsq.fr](mailto:stephane.lopes@uvsq.fr)



2016–2017

# Chapitre I : Preamble

1 Objectifs et prérequis

2 Plan

# Objectifs du cours

## Se focaliser sur les concepts de la POO

Aborder la programmation objet en se focalisant sur les concepts (définition, application) et non pas sur la syntaxe spécifique à un langage.

- Compréhension des concepts objets
- Mise en œuvre de ces concepts avec le langage Java
- Montrer l'importance des *API* (*Application Programming Interfaces*) et des bibliothèques

# Prérequis

- Notions de base en algorithmique
- Connaissance de base d'un langage impératif (C, ...)

# Plan général

- Introduction
- Vue d'ensemble des concepts objets
- Classe
- Héritage
- Module
- Gestion d'erreurs
- Entrées/sorties
- Collections
- Interfaces graphiques

## Chapitre II : Introduction

- 3 Caractéristiques des langages de programmation
- 4 Évolution vers les objets
- 5 La technologie Java
- 6 Le langage Java

# Chapitre II : Introduction

## Section 3 : Caractéristiques des langages de programmation

# Langage compilé vs. interprété

- Avec un *langage compilé*, le code source du programme est transformé en code machine par le compilateur
- Dans un *langage interprété*, le code source du programme est exécuté « à la volée » par l'interpréteur
- Certains langages sont à la fois compilés et interprétés



# Langage impératif vs. déclaratif

## Langage impératif

- Un *langage impératif* représente un programme comme une séquence d'instructions qui modifient son état au cours de son exécution
- Un programme décrit **comment** aboutir à la solution du problème
- Proche de l'architecture matérielle des ordinateurs (*architecture de von Neumann*)

## Langage déclaratif

- Un *langage déclaratif* permet de décrire ce que le programme doit faire (le **quoi**) et non pas comment il doit le faire (le **comment**)
- Un programme respectant ce style décrit le problème à traiter.

# Système de typage

- Un *système de typage* attribue des types aux éléments du langage
- Le typage est
  - explicite** si les types apparaissent dans le code source
  - implicite** si les types sont déterminés par le compilateur (*inférence de type*)
  - fort** si les manipulations entre données de types différents sont interdites
  - faible** si les possibilités de « trans-typage » sont nombreuses
  - statique** si la vérification des types est réalisée à la compilation
  - dynamique** si la vérification se fait à l'exécution

# Support des paradigmes de programmation

## Paradigme

Un paradigme de programmation représente la façon d'aborder un problème et d'en concevoir la solution.

- Programmation impérative
  - Programmation structurée
  - Programmation modulaire
  - Programmation par abstraction de données
  - Programmation objet
- Programmation fonctionnelle
- Programmation logique
- ...

## Support d'un paradigme

Un langage supporte un paradigme quand il fournit les fonctionnalités pour utiliser ce style (de façon simple, sécurisée et efficace)

# Exemple

## Programmation logique avec Prolog

- Prolog permet de définir et d'interroger une *base de faits*
- Prolog est un langage déclaratif
- Un *fait* est une assertion simple (*Idéfix est un chien.*)
- Une *règle* décrit une inférence à partir des faits (*Les chiens aiment les arbres*)
- Une *requête* est une question sur la *base de connaissance* (*Idéfix aime-t'il les arbres ?*)

# Exemple - Solveur de Sudoku 4 × 4 en Prolog

## Requête

```
- requête  
| ?- sudoku([_, _, 2, 3,  
            _, _, _, _  
            3, 4, _, _],  
            Solution).
```

Listing 1 – Requête en Prolog (Sudoku)

# Exemple - Solveur de Sudoku $4 \times 4$ en Prolog

## Résolution (partie 1)

- *la solution doit être unifiée avec le problème*
- *le problème comporte 16 chiffres*
- *chaque chiffre est compris entre 1 et 4 (fd\_domain)*

```
sudoku(Puzzle, Solution) :-
    Solution = Puzzle,
    Puzzle = [A1, A2, A3, A4,
              B1, B2, B3, B4,
              C1, C2, C3, C4,
              D1, D2, D3, D4],
    fd_domain(Puzzle, 1, 4),
```

## Listing 2 – Résolution du Sudoku (partie 1)

# Exemple - Solveur de Sudoku $4 \times 4$ en Prolog

## Résolution (partie 2)

– *les blocs (lignes, colonnes et carrés) sont définis*

```
Row1 = [A1, A2, A3, A4],
```

```
Row2 = [B1, B2, B3, B4],
```

```
Row3 = [C1, C2, C3, C4],
```

```
Row4 = [D1, D2, D3, D4],
```

```
Col1 = [A1, B1, C1, D1],
```

```
Col2 = [A2, B2, C2, D2],
```

```
Col3 = [A3, B3, C3, D3],
```

```
Col4 = [A4, B4, C4, D4],
```

```
Square1 = [A1, A2, B1, B2],
```

```
Square2 = [A3, A4, B3, B4],
```

```
Square3 = [C1, C2, D1, D2],
```

```
Square4 = [C3, C4, D3, D4],
```

## Listing 3 – Résolution du Sudoku (partie 2)

# Exemple - Solveur de Sudoku $4 \times 4$ en Prolog

## Résolution (partie 3)

- le prédicat *valid* reçoit une liste de 12 blocs
- la liste vide est valide
- la tête de la liste ne comporte pas de doublons (*fd\_all\_different*)
- le reste de la liste doit être valide

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).

– une solution possède des blocs valides

valid([Row1, Row2, Row3, Row4,
      Col1, Col2, Col3, Col4,
      Square1, Square2, Square3, Square4]).
```

## Listing 4 – Résolution du Sudoku (partie 3)



# Exemple

## Programmation fonctionnelle avec Haskell

- Haskell est un langage fonctionnel
- Possède un système de typage statique, fort et principalement implicite (inférence de types)

# Exemple - Haskell (partie 1)

```
— Calcul de la fonction factorielle

— Récursive
fact x = if x == 0 then 1 else fact (x - 1) * x

— Pattern matching
fact 0 = 1
fact x = x * fact (x - 1)

— Gardes
fact x
| x > 1 = x * fact (x - 1)
| otherwise = 1

— Liste et intervalle
fac x = product [1..x]
```

Listing 5 – Factoriel avec Haskell

# Exemple - Haskell (partie 2)

— Fonctions d'ordre supérieure

```
mapList f [] = []
```

```
mapList f (x:xs) = f x : mapList f xs
```

— Listes en compréhension et évaluation paresseuse

```
take 10 [ (i,j) | i <- [1..], j <- [1..], i < j ]
```

## Listing 6 – Exemples en Haskell

# Chapitre II : Introduction

## Section 4 : Évolution vers les objets

- Évolution des paradigmes impératifs
- Difficultés de l'approche objet
- Quelques langages de programmation

# Motivations

- Un logiciel est difficile à développer, à modifier et à maintenir
- La plupart des logiciels sont livrés en retard et dépassent leur budget
- Les programmeurs « réinventent souvent la roue » car il y a peu de *réutilisation* de code

⇒ Nécessité de trouver une approche plus efficace

# Chapitre II : Introduction

## Section 4 : Évolution vers les objets

- Évolution des paradigmes impératifs
- Difficultés de l'approche objet
- Quelques langages de programmation

# Programmation structurée

## Definition

*Choisissez les procédures. Utiliser les meilleurs algorithmes que vous pourrez trouver.*

## Caractéristiques

- L'accent est mis sur les *traitements*
- Approche très utilisée depuis de nombreuses années
- Approche de haut en bas (*top-down*)
- A beaucoup amélioré la qualité du logiciel

## Limitations

*Les données et les traitements restent indépendants.*

# Programmation modulaire

## Definition

*Choisissez vos modules. Découpez le programme de telle sorte que les données soient masquées par ces modules.*

## Caractéristiques

- Un module est un ensemble de procédures connexes avec les données qu'elles manipulent
- L'élément primordial passe de la conception des procédures à l'organisation des données
- Principe de *masquage de l'information*
- Permet la *compilation séparée*

## Limitations

*Les types ainsi définis diffèrent dans leur utilisation (création, ...) des types de base.*



# Programmation par abstraction de données

## Definition

*Choisissez les types dont vous avez besoin. Fournissez un ensemble complet d'opérations pour chaque type.*

## Caractéristiques

- Utilise les *types abstraits de données* (TAD)
- L'*interface* d'un type abstrait isole complètement l'utilisateur des détails d'implémentation

## Limitations

*Il est nécessaire de modifier un type pour l'adapter.*

# Programmation orientée objet

## Definition

*Choisissez vos classes. Fournissez un ensemble complet d'opérations pour chaque classe. Rendez toute similitude explicite à l'aide de l'héritage.*

## Caractéristiques

- Consiste à définir les classes puis à préciser les relations entre elles (notamment l'héritage)

# Chapitre II : Introduction

## Section 4 : Évolution vers les objets

- Évolution des paradigmes impératifs
- Difficultés de l'approche objet
- Quelques langages de programmation

# Approche objet vs. approche structurée

- **L'approche objet est moins intuitive**
  - décomposer un problème en une hiérarchie de fonctions atomiques et de données est plus naturel
- **La modélisation objet est difficile**
  - Rien dans les concepts de base de l'approche objet ne dicte comment modéliser la structure objet d'un système de manière pertinente.
  - Comment mener une analyse qui respecte les concepts objet ?
  - Sans un cadre méthodologique approprié, la dérivation fonctionnelle de la conception est inévitable.
- **L'application des concepts objet nécessite de la rigueur**
  - Le vocabulaire précis est un facteur d'incompréhensions.

# Approche objet vs. langage de programmation

- Beaucoup de développeurs ne pensent objet qu'à travers un langage de programmation
- Les langages ne sont que des outils implémentant certains concepts objet d'une certaine façon
- Les langages ne garantissent en rien l'utilisation adéquat de ces moyens techniques

## Programmer en Java ou en C# n'est pas concevoir objet !

- Seule une analyse objet conduit à une solution objet, i.e. qui respecte les concepts de base de l'approche objet.
- Le langage de programmation est un moyen d'implémentation
- Il ne garantit pas le respect des concepts objet

# Conception objet

## Definition

Concevoir objet, c'est d'abord concevoir un *modèle* qui respecte les concepts objet.

- Pour penser et concevoir objet, il faut savoir « prendre de la hauteur », jongler avec des concepts abstraits, indépendants des langages d'implémentation et des contraintes purement techniques
- Les langages de programmation ne sont pas un support adéquat pour cela
- Pour conduire une analyse objet cohérente, il ne faut pas directement penser en terme de pointeurs, d'attributs et de tableaux, mais en terme d'association, de propriétés et de cardinalités

# Chapitre II : Introduction

## Section 4 : Évolution vers les objets

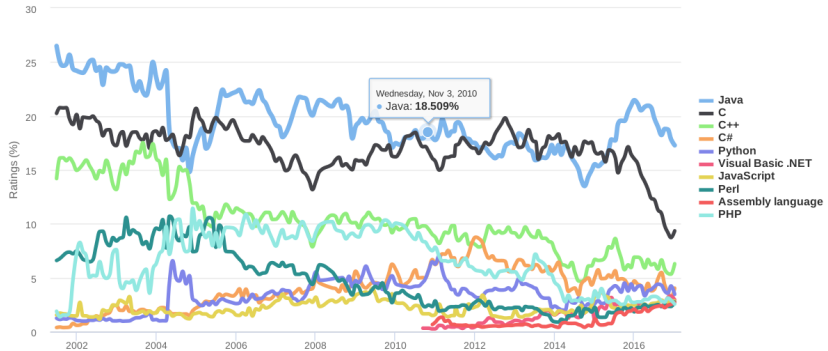
- Évolution des paradigmes impératifs
- Difficultés de l'approche objet
- Quelques langages de programmation

# Les langages OO « historiques »

- Simula, Ole-Johan Dahl et Kristen Nygaard (années 1960)
- SmallTalk, Alan Kay et Dan Ingalls (début des années 1970)
- Eiffel, Bertrand Meyer (1985)
- C++, Bjarne Stroustrup (années 1980, Standard ISO en 1998)



# Les principaux langages actuels I

























source : <http://www.tiobe.com/tiobe-index/>

# Les principaux langages actuels II



source : <http://pypl.github.io/PYPL.html>

# Les principaux langages actuels III

Language Rank	Types	Spectrum Ranking
1. C	  	100.0
2. Java	  	98.1
3. Python	 	98.0
4. C++	  	95.9
5. R		87.9
6. C#	  	86.7
7. PHP		82.8
8. JavaScript	 	82.2
9. Ruby	 	74.5
10. Go	 	71.9

source : <http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages>

# Les principaux langages actuels IV

- C, Denis Ritchie (1972)
- C#, Anders Hejlsberg (2001)
- Java, James Gosling (1995)
- Javascript, Brendan Eich (1995)
- PHP, Rasmus Lerdorf (1994)
- Python, Guido van Rossum (1990)

# Les langages « tendances »

- [Go](#), Robert Griesemer, Rob Pike, Ken Thompson (2009)
- [Groovy](#), Java Community Process (2003)
- [Julia](#), Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman (2012)
- [R](#), Ross Ihaka, Robert Gentleman (1993)
- [Rust](#), Graydon Hoare (2010)
- [Scala](#), Martin Odersky (2003)
- [Swift](#), Apple (2014)

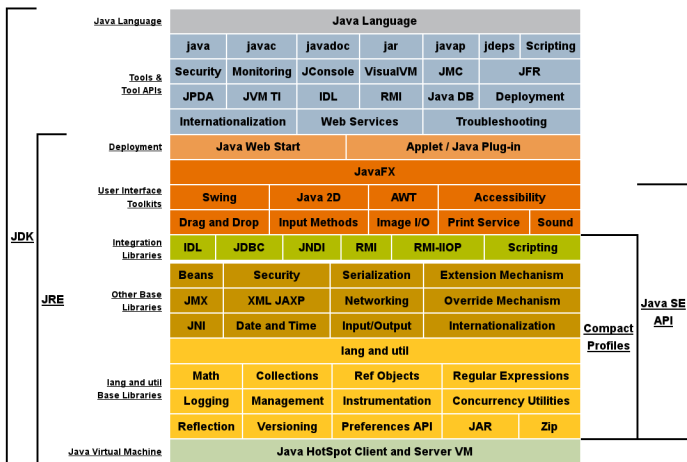
# Chapitre II : Introduction

## Section 5 : La technologie Java

# Les plateformes Java

- Java Platform Standard Edition (Java SE)
  - dédiée aux postes clients et aux stations de travail
- Java Platform Enterprise Edition (Java EE)
  - dédiée aux applications d'entreprises (serveur, postes clients, ...)
- Java Embedded
  - dédiée aux systèmes embarqués (Internet des Objets, ...)

# Vue d'ensemble de la plateforme Java SE



source : [Java Platform SE 8 Documentation](#)



# Java Runtime Environment

- Le *Java Runtime Environment* (JRE) fournit la *machine virtuelle Java*, les bibliothèques et d'autres composants nécessaires pour l'exécution de programmes Java
- Déjà installé sur la plupart des systèmes d'exploitation
- Le lanceur d'application (`java`) est l'outil ligne de commande permettant l'exécution de programme Java

## Plus d'informations

[Java Platform Overview](#)

# Java Development Kit

- Le *Java Development Kit* (JDK) fournit le JRE ainsi qu'un ensemble d'outils pour le développement d'applications
- **Doit être installé pour développer en Java**
- L'outil `javac` est le compilateur en ligne de commande

## Plus d'informations

[Java Platform Overview](#)

# Chapitre II : Introduction

## Section 6 : Le langage Java

- Caractéristiques du langage
- Constructions de base du langage Java

# Chapitre II : Introduction

## Section 6 : Le langage Java

- Caractéristiques du langage
- Constructions de base du langage Java

# Caractéristiques du langage

**Simple** un développeur peut être rapidement opérationnel

**Orienté-objet** langage relativement «pur»

**Interprété** la compilation génère un code intermédiaire qui est interprété

**Portable** un programme compilé fonctionne sans modification sur différentes plateformes

**Robuste** vérifications à la compilation et à l'exécution

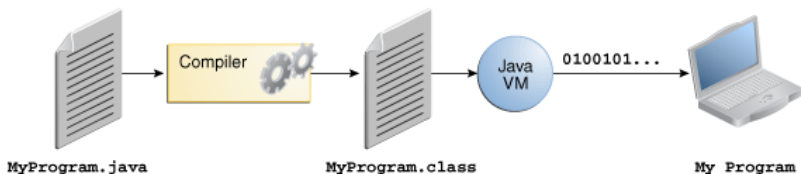
**Multithread** supporte la programmation concurrente

**Adaptable** supporte le chargement dynamique de code

**Sécurisé** le langage intègre un modèle de sécurité sophistiqué

# Compilation et interprétation

- Le langage Java est à la fois *interprété* et *compilé*
- Un fichier source (`.java`) est compilé en un langage intermédiaire appelé *bytecode* (`.class`)
- Ce bytecode est ensuite interprété par la *machine virtuelle Java*



source : *About the Java Technology*

# Compilation en ligne de commande (JDK)

```
$ javac <options> <fichiers source>
```

- g|g:none gère les informations pour le débogage
- classpath|-cp fixe le chemin de recherche des classes compilées (*Classpath*)
- source précise la version des fichiers sources (1.6, ..., 1.8)
- sourcepath fixe le chemin de recherche des sources
- encoding précise l'encodage des fichiers sources ("UTF-8", ...)
- d fixe le répertoire de destination pour les classes compilées
- target précise la version de la VM cible

## Compilation séparant les sources des fichiers compilés

```
$ javac -sourcepath src -source 1.7 \  
    -d classes -classpath classes \  
    -g src/MonApplication.java
```

# Exécution en ligne de commande (JRE)

```
$ java [-options] class [args...]  
$ java [-options] -jar jarfile [args...]
```

`class` est ici le nom de la classe (le `.class` doit pouvoir être trouvé dans le `CLASSPATH`)

`-cp|-classpath` fixe le chemin de recherche des classes compilées

`-jar` exécute un programme encapsulé dans un fichier jar

## Exécution

```
$ java -cp classes MonApplication
```



# Le *Classpath*

- Le *Classpath* précise la liste des bibliothèques ou des classes compilées utilisées par l'environnement Java
- Le compilateur ou la machine virtuelle ont besoin d'avoir accès aux classes compilées
- Il peut être défini en ligne de commande ou par la variable d'environnement CLASSPATH

# Chapitre II : Introduction

## Section 6 : Le langage Java

- Caractéristiques du langage
- Constructions de base du langage Java

# Notions de base

## Syntaxe

- Java différencie majuscules et minuscules
- Java possède une syntaxe proche du C

## Commentaires

`/* ... */` le texte entre `/*` et `*/` est ignoré

`// ...` le texte jusqu'à la fin de la ligne est ignoré

# Types primitifs

## Type primitif

Un *type primitif* est un type de base du langage, i.e. non défini par l'utilisateur. En Java, les valeurs de ces types ne sont pas des objets.

`boolean` true ou false

`byte` entier de  $-128$  à  $127$  (les types entiers sont signés)

`short` entier de  $-32\,768$  à  $32\,767$

`int` entier de  $-2^{31}$  à  $2^{31} - 1$

`long` entier de  $-2^{63}$  à  $2^{63} - 1$

`char` caractère Unicode sur 16 bits de `'\u0000'` à `'\uffff'`

`float` nombre en virgule flottante simple précision (32 bits IEEE 754)

`double` nombre en virgule flottante double précision (64 bits IEEE 754)

# Littéraux

## Littéral

Un *littéral* est la représentation dans le code source d'une valeur d'un type.

**Entiers** 123 de type `int`, 123L de type `long`, 0x123 en hexadécimal, 0b101 en binaire (**Java SE 7**)

**Flottants** 1.23E-4 de type `double`, 1.23E-4F de type `float`

**Booléens** `true` ou `false`

**Caractères** `'a'`, `'\t'` ou `'\u0000'`

**Chaînes** `"texte"`

**Null** `null`

# Exemple

## Déclarations et initialisations de variables

```
// Exemples de declaration avec initialisation
// (pas indispensable mais conseille)

byte aByte = 12;           // Un entier sur 8 bits
short aShort = 130;        // Un entier sur 16 bits
int aInteger = -153456;     // Un entier sur 32 bits

// Remarquer le L pour le litteral de type long
// (sinon erreur a la compilation: entier trop grand)
long aLong = 987654321234L; // Un entier sur 64 bits

// Remarquer le F pour le litteral de type float
// (sinon erreur a la compilation: perte de precision)
float aFloat = 1.3F;        // Un reel simple precision
double aDouble = -1.5E-4;   // Un reel double precision

char aChar = 'S';           // Un caractere
boolean aBoolean = true;    // Un booleen

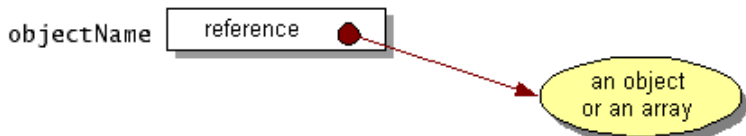
// La constante est introduite par le mot-cle final
final int aConst = 0;       // Une constante
```

### Listing 7 – Déclarations et initialisations de variables

# Références 1/2

- Les variables de type tableau, énumération, objet ou interface sont en fait des *références*
- La valeur d'une telle variable est une référence vers (l'adresse de) une donnée
- Dans d'autres langages, une référence est appelée *pointeur* ou *adresse mémoire*
- En Java, la différence réside dans le fait qu'on ne manipule pas directement l'adresse mémoire : le nom de la variable est utilisé à la place
- L'association (l'affectation) d'une donnée à une variable *lie* l'identificateur et la donnée

# Références 2/2



source : *The Java Tutorial*



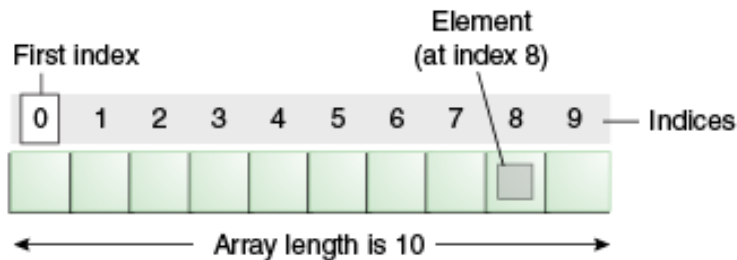
# Référence vs. pointeur

- Dans les deux cas, ce sont des variables (ou des constantes) dont la valeur (le contenu) est une adresse mémoire
- Un pointeur est un concept de bas-niveau permettant une manipulation précise de l'adresse (arithmétique des pointeurs, pointeur de fonction, ...)
- Une référence est une *abstraction* de plus haut niveau qui fournit une interface plus simple mais plus limitée pour manipuler l'adresse

Fonctionnalités	Référence	Pointeur
Simplicité	Oui	Non
Notation spécifique	Non	Oui
Arithmétique des adresses	Non	Oui
Adresse de fonctions	Non	Oui

# Tableaux

- Un *tableau* est une structure de données regroupant plusieurs valeurs de même type
- La taille d'un tableau est déterminée lors de sa création (à l'exécution)
- La taille d'un tableau ne varie pas par la suite
- Un tableau peut contenir des références



source : [The Java Tutorials: Arrays](#)

# Déclaration et création de tableaux

- La déclaration d'une variable de type tableau se fait en ajoutant `[]` au type des éléments

```
int [] unTableau;
```

- La création du tableau se fait en utilisant l'opérateur `new` suivi du type des éléments du tableau et de sa taille entre `[]`

```
new int [10];
```

- La référence retournée par `new` peut être liée à une variable

```
int [] unTableau = new int [10];
```

- Il est possible de créer et d'initialiser un tableau en une seule étape

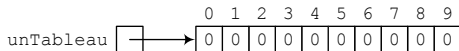
```
int [] unTableau = { 1, 5, 10 };
```

# Manipulation de tableaux

- L'accès aux éléments d'un tableau se fait en utilisant le nom du tableau suivi de l'indice entre `[]` (exemple : `unTableau[2]`)
- La taille d'un tableau peut être obtenue en utilisant la propriété `length` (exemple : `unTableau.length`)

# Tableaux et références

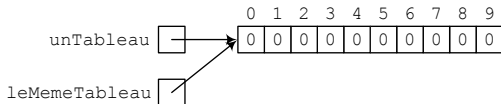
```
int [] unTableau = new int[10];
```



# Tableaux et références

```
int [] unTableau = new int[10];
```

```
int [] leMemeTableau = unTableau;
```

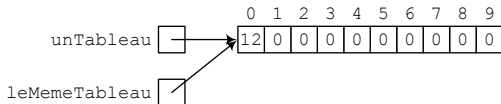


# Tableaux et références

```
int [] unTableau = new int[10];
```

```
int [] leMemeTableau = unTableau;
```

```
leMemeTableau[0] = 12;
```



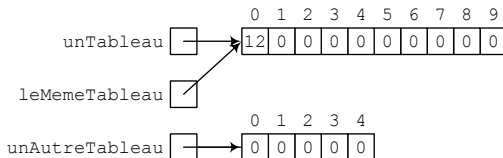
# Tableaux et références

```
int [] unTableau = new int[10];
```

```
int [] leMemeTableau = unTableau;
```

```
leMemeTableau[0] = 12;
```

```
int [] unAutreTableau = new int[5];
```





# Tableaux et références

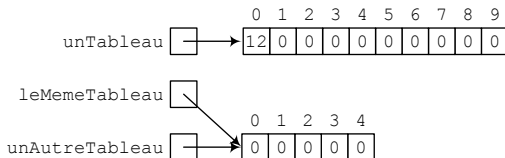
```
int [] unTableau = new int[10];
```

```
int [] leMemeTableau = unTableau;
```

```
leMemeTableau[0] = 12;
```

```
int [] unAutreTableau = new int[5];
```

```
leMemeTableau = unAutreTableau;
```



# Tableaux et références

```
int [] unTableau = new int[10];
```

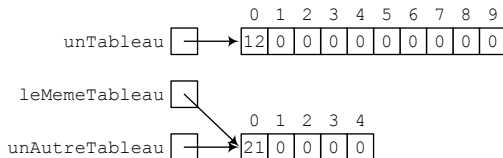
```
int [] leMemeTableau = unTableau;
```

```
leMemeTableau[0] = 12;
```

```
int [] unAutreTableau = new int[5];
```

```
leMemeTableau = unAutreTableau;
```

```
leMemeTableau[0] = 21;
```



## Chapitre III : Vue d'ensemble des concepts objets

7 Objet et message

8 Classe

9 Héritage

10 Module

# Chapitre III : Vue d'ensemble des concepts objets

## Section 7 : Objet et message

# Système orienté objet

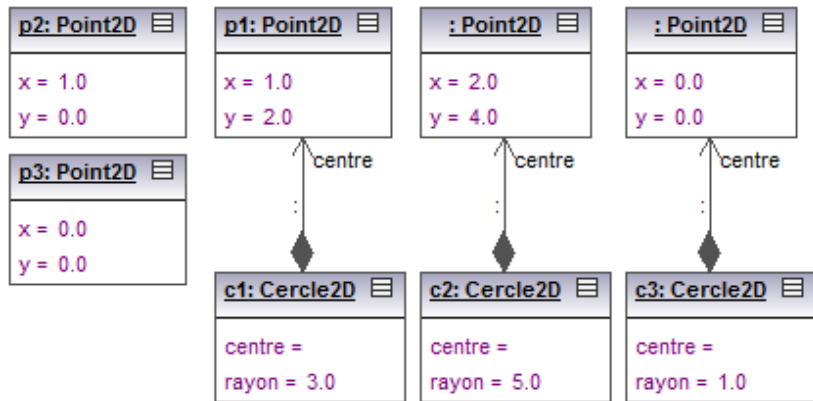
- Lors de son exécution, un *système OO* est **un ensemble d'objets qui interagissent**
- Les objets forment donc l'*aspect dynamique* (à l'exécution) d'un système OO
- Ces objets représentent soit
  - des entités du monde réel ( $\Rightarrow$  ce sont donc des modèles d'entités réelles), soit
  - des objets « techniques » nécessaires durant l'exécution du programme.

# Objet

- Un *objet* est formé de deux composants indissociables
  - son *état*, i.e. les valeurs prises par des variables le décrivant (*propriétés*)
  - son *comportement*, i.e. les opérations qui lui sont applicables
- Un objet est une *instance* d'une *classe*
- Un objet peut avoir plusieurs types, i.e. supporter plusieurs interfaces

# Exemple

## Des points et des cercles



# Exemple

## Des points et des cercles en Java

```
Point2D p1 = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(1.0);
Point2D p3 = new Point2D();
Point2D unAutreP3 = p3;
assert p3 == unAutreP3; // 2 points identiques

Cercle2D c1 = new Cercle2D(p1, 3.0);
Cercle2D c2 = new Cercle2D(new Point2D(2.0, 4.0), 5.0);
Cercle2D c3 = new Cercle2D();
```

Instanciation de cercles et de points en Java



# Communication par messages

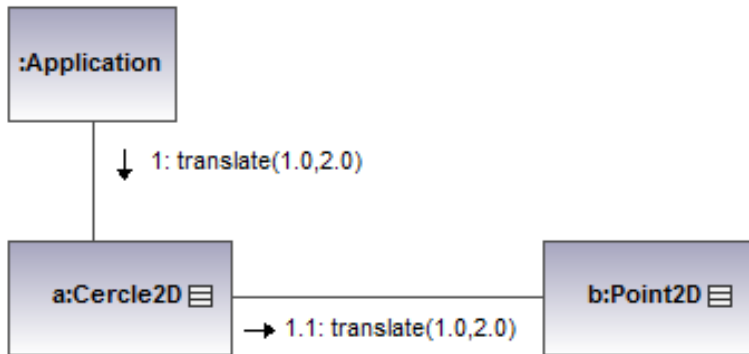
- Un objet solitaire n'a que peu d'intérêt  $\Rightarrow$  différents objets doivent pouvoir interagir
- Un *message* est un moyen de communication (d'interaction) entre objets
- **Les messages sont les seuls moyens d'interaction entre objets**  $\Rightarrow$  l'état interne ne doit pas être manipulé directement
- Le (ou les) type(s) d'un objet détermine les messages qu'il peut recevoir

# Message

- Un message est une requête envoyée à un objet pour demander l'exécution d'une opération
- Un message comporte trois composants :
  - l'objet auquel il est envoyé (le destinataire du message),
  - le nom de l'opération à invoquer,
  - les paramètres effectifs.

# Exemple

## Échange de messages



# Exemple

## Échange de messages en Java

- lors de l'exécution d'une opération de l'application, envoi de `a.translate(1.0, 2.0)`
  - lors de l'exécution de `translate` du cercle `a`, envoi de `b.translate(1.0, 2.0)`
    - exécution de `translate` du point `b`

# Chapitre III : Vue d'ensemble des concepts objets

## Section 8 : Classe

- Classe
- Classe et objet
- Classe et type

# Chapitre III : Vue d'ensemble des concepts objets

## Section 8 : Classe

- Classe
- Classe et objet
- Classe et type

# Classe I

- Une *classe* est un « modèle » (un « moule ») pour une catégorie d'objets structurellement identiques
- Une classe définit donc l'implémentation d'un objet (son état interne et le codage de ses opérations)
- L'ensemble des classes décrit l'*aspect statique* d'un système OO

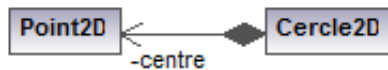
# Classe II

- Une classe comporte :
  - la définition des *attributs* (ou *variables d'instance*),
  - la *signature* des opérations (ou *méthodes*),
  - la *réalisation* (ou *définition*) des méthodes.
- Chaque instance aura sa propre copie des attributs
- La signature d'une opération englobe son nom et le type de ses paramètres
- L'ensemble des signatures de méthodes représente l'interface de la classe (publique)
- L'ensemble des définitions d'attributs et de méthodes forme l'implémentation de la classe (privé)



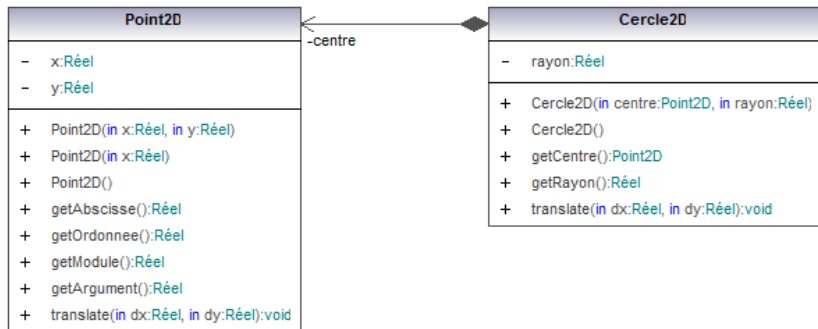
# Exemple

Les classes Cercle2D et Point2D (mode abrégé)



# Exemple

## Les classes Cercle2D et Point2D



# Exemple

## La classes Cercle2D en Java

```
package fr.uvsq.info.poo.coo;

public class Cercle2D implements Deplacable {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre
     * @param rayon Le rayon
     */
    public Cercle2D(Point2D centre, double rayon) { /* ... */ }

    /**
     * Initialise un cercle centré à l'origine et de rayon 1.
     */
    public Cercle2D() { /* ... */ }

    public Point2D getCentre() { return centre; }
    public double getRayon() { return rayon; }

    /**
     * Translate le cercle.
     * @param dx déplacement en abscisse.
     * @param dy déplacement en ordonnées.
     */
}
```

# Chapitre III : Vue d'ensemble des concepts objets

## Section 8 : Classe

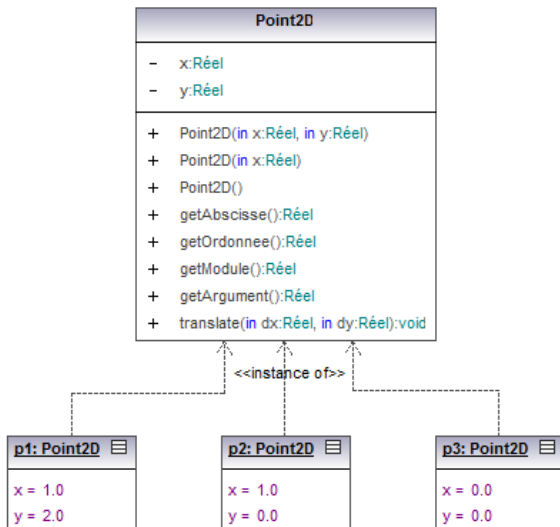
- Classe
- Classe et objet
- Classe et type

# Instanciation d'une classe

- Le mécanisme d'*instanciation* permet de créer des objets à partir d'une classe
- Chaque objet est une instance d'une classe
- Lors de l'instanciation,
  - de la mémoire est allouée pour l'objet,
  - l'objet est initialisé (appel du constructeur) afin de respecter l'invariant de la classe.

# Exemple

## Classe et objet



# Chapitre III : Vue d'ensemble des concepts objets

## Section 8 : Classe

- Classe
- Classe et objet
- Classe et type

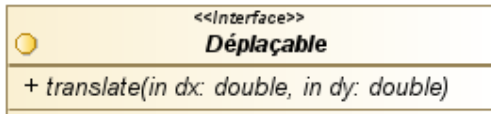
# Type

- Un *type* est un modèle abstrait réunissant à un haut degré les traits essentiels de tous les êtres ou de tous les objets de même nature
- En informatique, un *type (de donnée)* spécifie :
  - l'ensemble des valeurs possibles pour cette donnée (définition en *extension*),
  - l'ensemble des opérations applicables à cette donnée (définition en *intention*).
- Un type spécifie l'*interface* par laquelle une donnée peut être manipulée



# Exemple

Représentation d'un type comme une interface



# Exemple

## L'interface Deplacable en Java

```
package fr.uvsq.info.poo.coo;

public interface Deplacable {
    /**
     * Translate l'objet.
     *
     * @param dx déplacement en abscisse
     * @param dy déplacement en ordonnées
     */
    void translate(double dx, double dy);
}
```

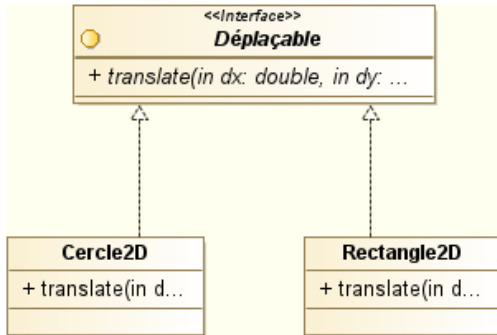
## L'interface Deplacable en Java

# Classe et type

- Une classe implémente un ou plusieurs types, i.e. respecte une ou plusieurs interfaces
- Un objet peut avoir plusieurs types mais est une instance d'une seule classe
- Des objets de classes différentes peuvent avoir le même type

# Exemple

## Interface et classe



```
public class Cercle2D implements Deplaçable {
```

Cercle2D implémente Deplaçable en Java

# Exercice I

## Modélisation d'objets et de classes

On veut modéliser des robots se déplaçant sur un terrain. Ce terrain est découpé en cases carrées repérées par deux coordonnées. Chaque case peut être vide ou contenir un mur ou un robot. Les robots sont très rudimentaires et ne disposent que d'une boussole. Ils ne connaissent donc que leur orientation (Nord, Est, Sud, Ouest). Un robot doit pouvoir avancer d'une case et tourner d'un quart de tour à droite. Un robot ne peut se déplacer d'une case à une autre que si la case de destination est vide.

# Exercice II

## Modélisation d'objets et de classes

- 1 Représenter avec la notation vue précédemment la classe Robot
- 2 Faire de même avec la classe Terrain
- 3 Représenter sur un diagramme de communication les échanges de messages pour le déplacement d'un robot
- 4 On suppose maintenant qu'un robot peut en détecter un autre qui passe devant lui. Par exemple, quand un robot se déplace, il peut passer dans le champ de vision d'un autre. Ce dernier devra alors être averti. Modifier le diagramme de communication précédent pour y intégrer la détection des déplacements

# Exercice (solutions)

## La classe Robot

Robot	
-	terrain:Terrain
-	orientation:Direction
+	Robot(in terrain:Terrain, in depart:Position, in orientation:Direction)
+	getOrientation():Direction
+	avance()
+	tourneADroite()

# Exercice (solutions)

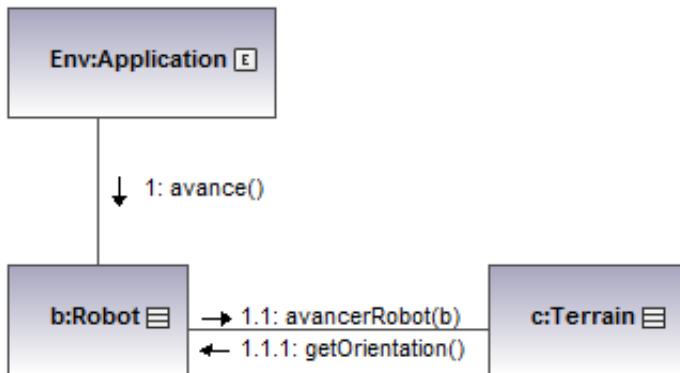
## La classe Terrain

Terrain	
-	terrain:Case[1..*]
+	Terrain(in tailleX, in tailleY, in murs:Position[*])
+	ajouterRobot(in robot:Robot, in orig:Position)
+	avancerRobot(in robot:Robot)



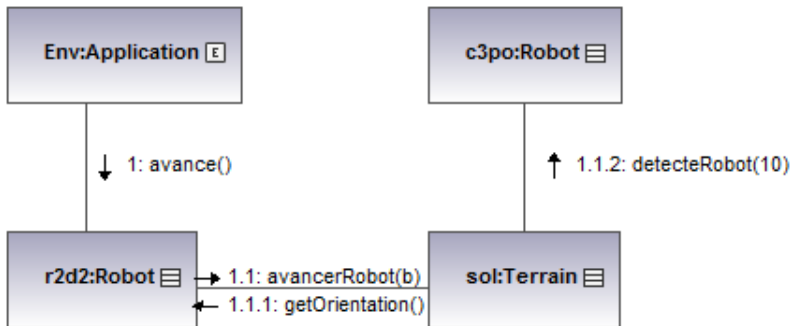
# Exercice (solutions)

## Déplacement d'un robot



# Exercice (solutions)

## Déplacement avec détection



- Ajout de la méthode `detecteRobot(distance : Entier)` dans `Robot`

# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

- Héritage
- Polymorphisme
- Classe abstraite
- Héritage multiple et à répétition
- Héritage et sous-typage

# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

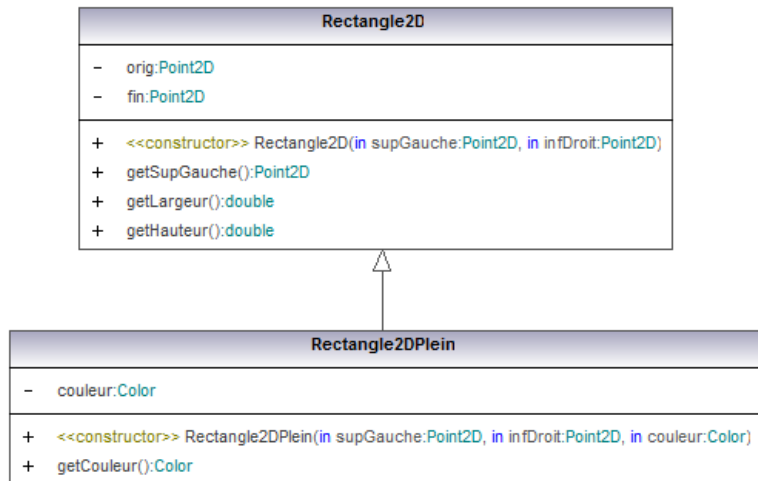
- Héritage
- Polymorphisme
- Classe abstraite
- Héritage multiple et à répétition
- Héritage et sous-typage

# Héritage

- L'*héritage* permet de définir l'implémentation d'une classe à partir de l'implémentation d'une autre
- Ce mécanisme permet, lors de la définition d'une nouvelle classe, de ne préciser que ce qui change par rapport à une classe existante
- Une *hiérarchie de classes* permet de gérer la complexité, en ordonnant les classes au sein d'arborescences d'abstraction croissante
- Si Y hérite de X, on dit que Y est une classe *filles* (*sous-classe*, *classe dérivée*) et que X est une classe *mère* (*super-classe*, *classe de base*)

# Exemple

## Rectangle et rectangle plein



# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

- Héritage
- Polymorphisme
- Classe abstraite
- Héritage multiple et à répétition
- Héritage et sous-typage

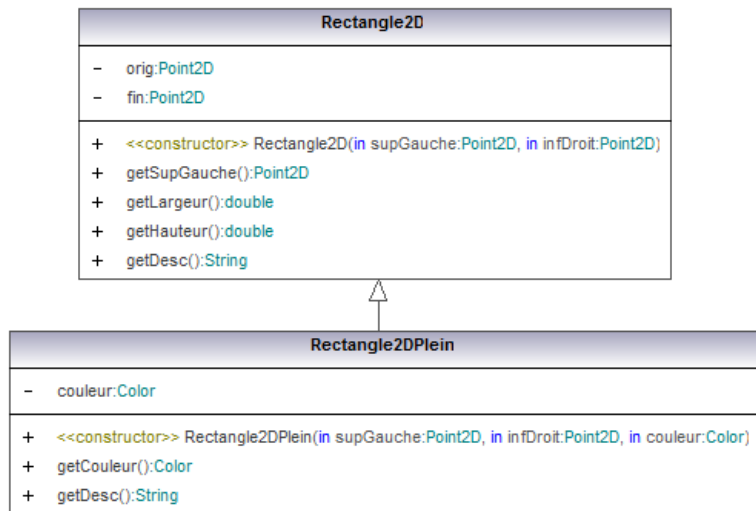
# Polymorphisme

- Le *polymorphisme* est l'aptitude qu'ont des objets à réagir différemment à un même message
- L'intérêt est de pouvoir gérer une collection d'objets de façon homogène tout en conservant le comportement propre à chaque objet
- Une méthode commune à une hiérarchie de classe peut prendre plusieurs formes dans différentes classes
- Une sous-classe peut *redéfinir* une méthode de sa super-classe pour spécialiser son comportement
- Le choix de la méthode à appeler est retardé jusqu'à l'exécution du programme (*liaison dynamique ou retardée*)



# Exemple

Une description pour le rectangle et le rectangle plein



# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

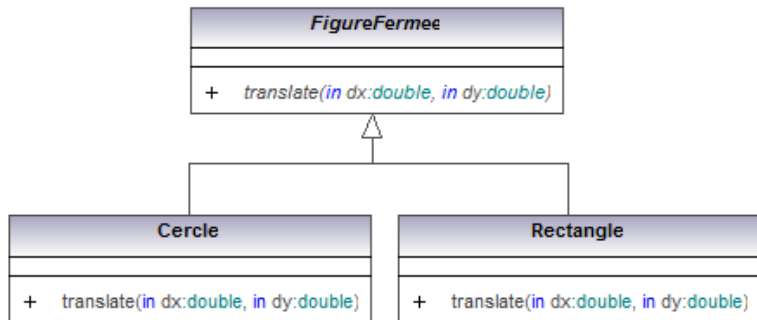
- Héritage
- Polymorphisme
- **Classe abstraite**
- Héritage multiple et à répétition
- Héritage et sous-typage

# Classe abstraite

- Une *classe abstraite* représente un concept abstrait qui ne peut pas être instancié
- En général, son comportement ne peut pas être intégralement implémenté à cause de son niveau de généralisation
- Elle sera donc seulement utilisée comme classe de base dans une hiérarchie d'héritage

# Exemple

## La hiérarchie d'héritage des figures



# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

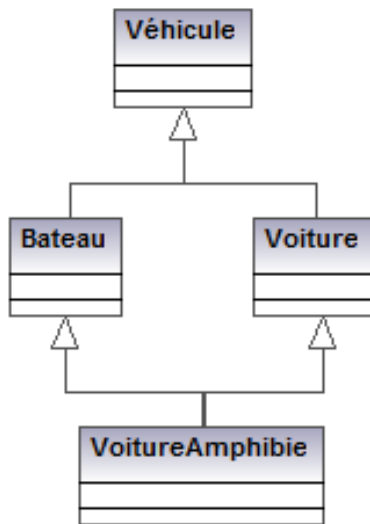
- Héritage
- Polymorphisme
- Classe abstraite
- Héritage multiple et à répétition
- Héritage et sous-typage

# Héritage multiple et à répétition

- Un *héritage multiple* se produit lorsqu'une classe possède plusieurs super-classes
- Un *héritage à répétition* se produit lorsqu'une classe hérite plusieurs fois d'une même super-classe
- Ces types d'héritage peuvent provoquer des conflits aux niveaux des attributs et méthodes
  - deux classes de base peuvent posséder la même méthode,
  - un attribut peut être hérité selon plusieurs chemins dans le graphe d'héritage.
- L'héritage multiple de classe n'est pas supporté par Java

# Exemple

## Héritage multiple et à répétition



# Chapitre III : Vue d'ensemble des concepts objets

## Section 9 : Héritage

- Héritage
- Polymorphisme
- Classe abstraite
- Héritage multiple et à répétition
- Héritage et sous-typage



# Sous-type

## Sous-type

Un type  $T_1$  est un *sous-type* d'un type  $T_2$  si l'interface de  $T_1$  contient l'interface de  $T_2$

- Un sous-type possède une interface plus riche, i.e. au moins toutes les opérations du super-type
- De manière équivalente, l'extension du super-type contient l'extension du sous-type, i.e. tout objet du sous-type est aussi instance du super-type

# Principe de substitution de Liskov

## Principe de substitution de Liskov

Si pour chaque objet  $o_1$  de type  $S$ , il existe un objet  $o_2$  de type  $T$  tel que, pour tout programme  $P$  défini en terme de  $T$ , le comportement de  $P$  demeure inchangé lorsque  $o_1$  est remplacé par  $o_2$ , alors  $S$  est un sous-type de  $T$ .

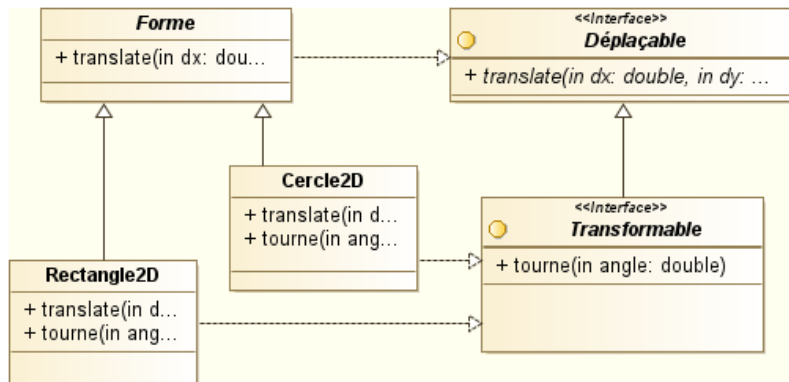
- Un objet du sous-type peut remplacer un objet du super-type sans que le comportement du programme ne soit modifié

# Héritage et sous-typage

- L'héritage (ou *héritage d'implémentation*) est un mécanisme technique de réutilisation
- Le sous-typage (ou *héritage d'interface*) décrit comment un objet peut être utilisé à la place d'un autre
- Si Y est une sous-type de X, cela signifie que « Y est une sorte de X » (relation *IS-A*)
- Dans un langage de programmation, les deux visions peuvent être représentées de la même façon : le mécanisme d'héritage permet d'implémenter l'un ou l'autre
- En Java, les interfaces et l'héritage entre interface implémentent plus spécifiquement le concept de sous-typage

# Exemple

## Héritage d'implémentation et d'interface



# Chapitre III : Vue d'ensemble des concepts objets

## Section 10 : Module

# Module

- Un *module* (ou *package*) est l'unité de base de décomposition d'un système
- Il permet d'organiser logiquement des modèles
- Un module s'appuie sur la notion d'*encapsulation*
  - publie une interface, i.e. ce qui est accessible de l'extérieur
  - utilise le principe de *masquage de l'information*, i.e. ce qui ne fait pas parti de l'interface est dissimulé

# Utilité d'un module

- Sert de brique de base pour la construction d'une architecture
- Représente le bon niveau de granularité pour la réutilisation
- Est un *espace de noms* qui permet de gérer les conflits

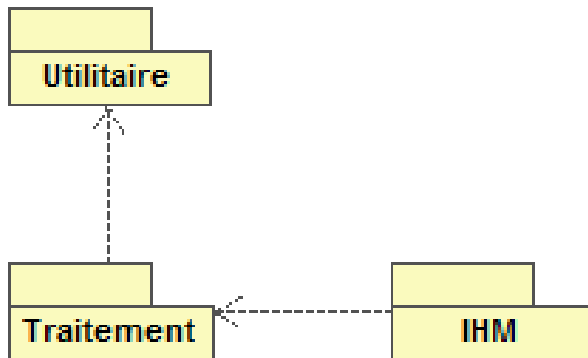
# Qualité d'un module

- La conception d'un module devrait conduire à un *couplage faible* et une *forte cohésion*
  - couplage** désigne l'importance des liaisons entre les éléments  $\Rightarrow$  **doit être réduit**
  - cohésion** mesure le recouvrement entre un élément de conception et la tâche logique à accomplir  $\Rightarrow$  **doit être élevé**, i.e. chaque élément est responsable d'une tâche précise



# Exemple

Un diagramme de packages UML



# Module en Java

- Le mot clé `package` placé en début de fichier permet l'ajout d'éléments dans un module
- Le mot clé `import` permet l'accès aux éléments d'un module

## Chapitre IV : Objet et classe

- 11 Compléments sur les objets
- 12 Compléments sur les classes
- 13 Métaclasse
- 14 Généricité

# Chapitre IV : Objet et classe

## Section 11 : Compléments sur les objets

- Caractéristiques d'un objet
- Les objets en Java
- Les chaînes de caractères en Java

# Chapitre IV : Objet et classe

## Section 11 : Compléments sur les objets

- Caractéristiques d'un objet
- Les objets en Java
- Les chaînes de caractères en Java

# Etat d'un objet

- L'état d'un objet est l'ensemble de ses caractéristiques **internes**, **cachées** aux autres objets
- Ces caractéristiques peuvent elles-mêmes être des objets
- Les propriétés représentant l'état d'un objet particulier sont appelées *variables d'instance* ou *attributs* ou *données membres*
- Chaque objet possède un état courant décrit par la valeur de ses attributs et donc sa propre copie des variables d'instance
- Les attributs conservent leurs valeurs durant toute la durée de vie d'un objet

# Comportement d'un objet

- Le comportement d'un objet regroupe les caractéristiques **externes** mises à la disposition des autres objets
- Chaque opération est décrite par sa *signature* (son nom, les objets qu'elle prend comme paramètre et le type de retour)
- L'ensemble de ces opérations forme l'interface de l'objet
- Les opérations propres à un objet sont appelées *méthodes d'instance* ou *fonctions membres* (fonctions associées à l'objet)
- Ces opérations sont invoquées par rapport à un objet particulier
- Les opérations sont les seuls moyens de manipuler l'état interne d'un objet (*encapsulation*)

# Identité, égalité, affectation et copie d'objets

- Un objet possède une *identité* (identifiant interne) qui caractérise son existence de façon indépendante de son état (*égalité*  $\neq$  *identité*)
  - ⇒ deux objets peuvent être égaux (même état interne) sans être identiques (même objet)
- L'*affectation* d'un objet à une variable crée un *lien* entre la variable et l'objet
- Créer une *copie* d'un objet nécessite de créer récursivement une copie de ses attributs (*copie profonde* ou *deep copy*)
- Un autre sémantique possible est de ne copier que l'objet racine et de partager ses attributs (*copie superficielle* ou *shallow copy*)



# Chapitre IV : Objet et classe

## Section 11 : Compléments sur les objets

- Caractéristiques d'un objet
- Les objets en Java
- Les chaînes de caractères en Java

# Création d'objets

- Un objet est créé (instancié) à partir d'une classe en utilisant le mot-clé `new` (`new Point2D(1.0, 2.0)`)
- L'utilisation de `new` provoque la réservation de mémoire pour l'objet et l'invocation du constructeur qui initialise l'objet
  - Le constructeur est une méthode spéciale invoquée automatiquement lors de la création de l'objet
  - Son rôle est d'initialiser l'objet pour que ce dernier respecte l'invariant de la classe
- `new` retourne une référence sur l'objet créé
- Cette référence doit être liée (affectée) à un identificateur (variable) pour permettre l'accès à l'objet


# Association d'une variable à un objet

- La syntaxe pour la déclaration d'une variable est `type nom (Point2D p1)`
- **La déclaration ne crée pas d'objet** mais uniquement une référence
- La déclaration crée un identificateur qui sera lié à l'objet en utilisant une affectation
- La variable est invalide tant qu'elle n'est pas liée à un objet (*null reference*)
- Il est possible de lier la variable lors de sa déclaration  
(`Point2D p1 = new Point2D(1.0, 2.0);`)

# Exemple

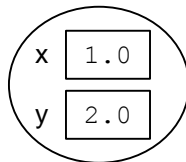
## Association d'une variable à un objet

```
Point2D p1;
```

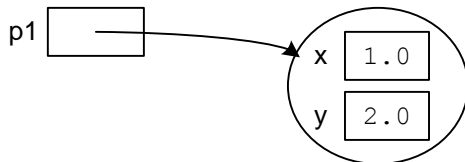
p1 

```
new Point2D(1.0, 2.0);
```

Instance de la classe Point2D



```
Point2D p1 = new Point2D(1.0, 2.0);
```



# Exemple

## Instanciation de cercle et de points 1/2

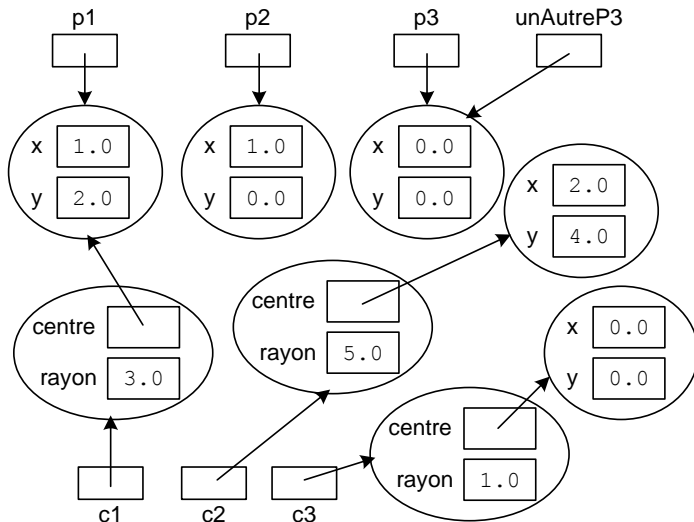
```
Point2D p1 = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(1.0);
Point2D p3 = new Point2D();
Point2D unAutreP3 = p3;
assert p3 == unAutreP3; // 2 points identiques

Cercle2D c1 = new Cercle2D(p1, 3.0);
Cercle2D c2 = new Cercle2D(new Point2D(2.0, 4.0), 5.0);
Cercle2D c3 = new Cercle2D();
```

## Instanciation de cercles et de points

# Exemple

## Instanciation de cercle et de points 2/2



# Manipulation

## Accès aux attributs

- Juste par le nom de l'attribut quand on se trouve dans la portée de l'attribut (exemple : `x`)
- En qualifiant/préfixant avec le nom d'une référence sur l'objet à l'extérieur (exemple : `p1.x`)
- **La manipulation directe d'attributs en dehors de la classe est interdite** (violation de l'encapsulation)

## Invocation d'une méthode

- Même syntaxe que pour les attributs mais avec la liste des paramètres (exemple : `p1.getAbscisse()`)

# Destruction

- Quand un objet n'est plus utilisé, il doit être retiré de la mémoire
- La destruction des objets en Java est automatique
  - L'environnement d'exécution de Java supprime les objets lorsqu'il détermine qu'ils ne sont plus utilisés
  - Un objet est éligible pour la destruction quand plus aucune référence n'est liée à lui
- Ce processus de suppression s'appelle *ramasse-miette* (*garbage collector*)
- Avant de détruire l'objet, le ramasse-miette invoque la méthode `protected void finalize ()` de l'objet
  - utilisée pour restituer les ressources allouées par l'objet
  - `finalize` est membre de la classe `Object`
  - `super.finalize()` doit être d'appeler à la fin de la méthode



# Chapitre IV : Objet et classe

## Section 11 : Compléments sur les objets

- Caractéristiques d'un objet
- Les objets en Java
- Les chaînes de caractères en Java

# Les chaînes de caractères en Java

- Java fournit trois classes pour les chaînes de caractères : `String`, `StringBuffer` et `StringBuilder`
- `String` est dédiée aux chaînes de caractères immuables, i.e. dont la valeur ne change pas
- `StringBuilder` est dédiée aux chaînes de caractères pouvant être modifiées (contexte *mono-thread*)
- `StringBuffer` est dédiée aux chaînes de caractères pouvant être modifiées (contexte *multi-threads*)

# Création d'une chaîne (String)

- Une instance de `String` représente une chaîne au format UTF-16
- Une chaîne est souvent créée à partir d'un littéral de type chaîne (une suite de caractères entre guillemets)
  - quand Java rencontre un littéral de type chaîne, il crée un objet de type `String` dont la valeur est le littéral
- Une chaîne peut aussi être créée en utilisant l'un des constructeurs de `String`

# Quelques accesseurs de String

`length()` taille de la chaîne,

`charAt(int)` caractère à l'indice spécifié,

`substring(int, int)` extraction d'une sous-chaîne,

`indexOf(...)`, `lastIndexOf(...)` recherche dans la chaîne

# Autres usages de String

- Un littéral chaîne peut être utilisé à tout endroit où un objet String peut l'être
  - ⇒ on peut invoquer des méthodes de String sur un littéral chaîne
- L'opérateur + permet de concaténer des objets de type String
  - c'est le seul opérateur surchargé pour un objet en Java
- Une chaîne peut être utilisée avec l'instruction switch (Java SE 7)

# Les classes `StringBuilder` et `StringBuffer`

- Les instances disposent à peu près des mêmes accesseurs que `String`
- Quelques mutateurs : `append(...)` (ajout de caractères), `delete(...)` (suppression de caractères), `insert(...)` (insertion de caractères)
- `StringBuilder` est optimisée pour un environnement *mono-thread*
- `StringBuffer` est à utiliser dans un contexte *multi-threads*

# Exemple

## Manipulation de chaînes de caractères

```
String source = "abcde";  
int len = source.length();  
StringBuilder dest = new StringBuilder(len);  
  
for (int i = (len - 1); i >= 0; —i) {  
    dest.append(source.charAt(i));  
}  
  
assert dest.toString().equals("edcba") : dest;
```

Inverser une chaîne

# Chapitre IV : Objet et classe

## Section 12 : Compléments sur les classes

- Caractéristiques d'une classe
- Les classes en Java



# Chapitre IV : Objet et classe

## Section 12 : Compléments sur les classes

- Caractéristiques d'une classe
- Les classes en Java

# Catégories de méthodes

**Accesseur** permet de consulter l'état d'un objet

**Mutateur** modifie l'état d'un objet

- doit préserver l'invariant

**Constructeur** initialise un objet afin de le placer dans un état cohérent

- établit l'invariant de la classe
- appelé automatiquement lors de la création d'un objet

**Destructeur** libère les ressources allouées par l'objet

- appelé automatiquement lors de la destruction de l'objet

# Surcharge de méthode

- La *surcharge* (ou *polymorphisme ad hoc*) d'une méthode consiste à définir plusieurs méthodes de même nom mais ayant des signatures différentes
- Une opération n'est donc plus simplement identifiée par son nom mais également par le nombre, l'ordre et le type de ses arguments
- Le choix de la méthode est résolu lors de la compilation

## Exemple d'application

- La surcharge est souvent utilisée pour définir plusieurs constructeurs de façon à initialiser les objets de différentes manières
- Une alternative plus souple consiste à appliquer un modèle de conception **FABRIQUE** (FACTORY)

# Accès aux membres

- Une classe peut contrôler l'accès à ses membres (données ou fonctions)
  - `privé` accès limité à la classe
  - `protégé` accès limité à la classe et à ses sous-classes
  - `module` accès limité au module (*package*) contenant la classe
  - `public` accès non limité

# Invariant de classe

## Invariant de classe

L'*invariant d'une classe* impose une contrainte sur l'état des instances de la classe. Chaque objet doit respecter les conditions imposées par l'invariant.

- L'invariant est établi lors de la création d'une instance
- Il doit être vérifié avant l'exécution d'une opération publique
- Chaque opération publique doit rétablir l'invariant avant de se terminer
- L'invariant peut être violé temporairement lors de l'exécution d'une opération
- Avec un langage de programmation, l'invariant peut être exprimé avec des assertions ou avec une construction spécifique

# Chapitre IV : Objet et classe

## Section 12 : Compléments sur les classes

- Caractéristiques d'une classe
- Les classes en Java

# Définition de classe

- La *définition* d'une classe comporte deux parties : la *déclaration* et le *corps* de la classe

```
/**
 * Un bref commentaire.
 * Un commentaire plus détaillé...
 * @version oct 2008
 * @author Prénom NOM
 */
class NomClasse { // Déclaration de la classe
    // Corps de la classe
}
```

- La déclaration précise au compilateur un certain nombre d'informations sur la classe (son nom, ...)
- Le corps de la classe contient les attributs et les méthodes (les *membres*) de la classe

# Attribut

- La déclaration d'un attribut spécifie son nom et son type

```
/** Description de l'attribut. */  
Type nom;  
/** Description de l'attribut. */  
final Type nom;
```

- L'initialisation des attributs se fait dans un constructeur
- `final` est optionnel et permet de déclarer un attribut qui ne pourra être affecté qu'une unique fois



# Pseudo-attribut `this`

- Chaque classe possède un attribut privé particulier nommé `this` qui référence l'objet courant
- Cette attribut est maintenu par le système et ne peut pas être modifié par le programmeur
- `this` n'est accessible que dans le corps de la classe
- Utilité
  - passer l'objet courant en paramètre d'une méthode
  - lever certaines ambiguïtés à propos des membres
  - invoquer un autre constructeur dans un constructeur

# Exemple

## Les attributs d'un cercle

```
/**
 * Un cercle en deux dimensions.
 *
 * @version   jan. 2015
 * @author    Stephane Lopes
 */
class Cercle2D {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private final double rayon;
```

## Les attributs d'un cercle

# Méthode

- La *définition* d'une méthode comporte deux parties : la *déclaration* et le *corps* (l'*implémentation*) de la méthode

```
/**
 * Brève description de la méthode.
 * Une description plus longue...
 * @param param1 description du paramètre
 * @param ...
 * @return description de la valeur de retour
 */
TypeRetour nomMethode(listeDeParametres) { // Déclaration
    // Corps de la méthode
}
```

- TypeRetour est le type de la valeur retournée ou void si aucune valeur n'est retournée
- Dans le corps de la méthode, on utilise l'opérateur `return` pour renvoyer une valeur
- Un constructeur a le même nom que sa classe et ne possède pas de type de retour

# Paramètre de méthode

- `listeDeParamètres` est une liste de déclarations de variables séparées par des virgules
  - un paramètre peut être vu comme une variable locale à la méthode
  - `final` peut préfixer la déclarations si le paramètre ne doit pas être modifié
- Le passage de paramètres se fait *par valeur*
  - la valeur d'un paramètre d'un type primitif ne peut pas être modifiée
  - la valeur d'une référence ne peut pas être modifiée mais l'objet référencé peut l'être (comme avec un pointeur en C)

# Exemple

## Les constructeurs de la classe Cercle2D

```
/**
 * Initialise un cercle avec un centre et un rayon.
 * @param centre Le centre.
 * @param rayon Le rayon.
 */
public Cercle2D(final Point2D centre, final double rayon) {
    this.centre = centre;
    this.rayon = rayon;
}

/**
 * Initialise un cercle centre a l'origine et de rayon 1
 */
public Cercle2D() {
    this(new Point2D(), 1.0);
}
```

## Les constructeurs du cercle

# Exemple

## Les accesseurs de la classe Cercle2D

```
/**
 * Renvoie le centre du cercle.
 * @return le centre du cercle.
 */
public Point2D getCentre() {
    return centre;
}

/**
 * Renvoie le rayon du cercle.
 * @return le rayon du cercle.
 */
public double getRayon() {
    return rayon;
}
```

Les accesseurs du cercle

# Exemple

## Le mutateur de la classe Cercle2D

```
/**
 * Translate le cercle.
 * @param dx déplacement en abscisse.
 * @param dy déplacement en ordonnées.
 */
public void translate(final double dx, final double dy) {
    centre.translate(dx, dy);
}
```

Le mutateur du cercle

# Contrôle d'accès aux membres en Java

- Le contrôle de l'accès aux membres permet de spécifier l'interface d'un classe
- Le niveau d'accès est précisé en ajoutant un mot-clé devant la déclaration du membre (attribut ou méthode)
- Il peut prendre l'une des valeurs `private`, `public`, `protected` ou être absent

Niveau	Classe	Paquetage	Sous-classe	En dehors
<code>private</code>	X			
<code>aucun</code>	X	X		
<code>protected</code>	X	X	X	
<code>public</code>	X	X	X	X

- La restriction d'accès s'applique au niveau de la classe et non pas de l'objet
- Les attributs sont déclarés `private` pour respecter l'encapsulation



# Exercice

## Manipulation d'objets et définition de classes

Pour cet exercice, on reprendra les spécifications obtenues lors de l'exercice précédent.

- 1 Soit le terrain suivant 

R N		M
M		R O

 (M = mur, R = robot, N = nord, O = ouest). Écrire les instructions Java permettant de créer ce terrain puis de déplacer le robot (0, 0) en (1, 1).
- 2 Écrire en Java la classe Robot qui modélise les robots.

# Exercice (solutions)

## Déplacement du robot

```
// Creation du terrain
Position[] murs = { new Position(0, 1), new Position(2, 0) };
Terrain terrain = new Terrain(3, 2, murs);

// Creation des robots
Robot r2d2 = new Robot(terrain, new Position(0, 0),
    Direction.NORD);
Robot c3po = new Robot(terrain, new Position(2, 1),
    Direction.OUEST);

// Mouvements
r2d2.tourneADroite(); // (0, 0) vers l'est
r2d2.avance();        // (1, 0) vers l'est
r2d2.tourneADroite(); // (1, 0) vers le sud
r2d2.avance();        // (1, 1) vers le sud
```

## Déplacement du robot

# Exercice (solutions)

## Classe Robot 1/2

```
public class Robot {  
    /** Terrain sur lequel evolue le robot. */  
    private Terrain terrain;  
  
    /** Orientation du robot. */  
    private Direction orientation;  
  
    /**  
     * Initialise le robot.  
     * @param terrain terrain sur lequel le robot va evoluer  
     * @param depart position d'origine du robot  
     * @param direction orientation du robot  
     */  
    public Robot(Terrain terrain, Position depart, Direction orientation) {  
        assert terrain != null;  
        this.terrain = terrain;  
        terrain.ajouterRobot(this, depart);  
        this.orientation = orientation;  
    }  
}
```

## Classe Robot

# Exercice (solutions)

## Classe Robot 2/2

```
/**
 * Retourne l'orientation du robot.
 * @return orientation du robot
 */
public Direction getOrientation() {
    return orientation;
}

/**
 * Demande au robot d'avancer d'une case.
 * @return true si l'operation a ete effectuee
 */
public boolean avance() {
    return terrain.avancerRobot(this);
}

/**
 * Fait faire au robot une rotation d'un quart de tour a droite.
 */
public void tourneADroite() {
    orientation = orientation.next();
}
}
```

## Classe Robot

# Chapitre IV : Objet et classe

## Section 13 : Métaclasse

- Métaclasse et membres de classe
- Membres de classe en Java
- Le programme principal en Java
- Énumération en Java

# Chapitre IV : Objet et classe

## Section 13 : Métaclasse

- Métaclasse et membres de classe
- Membres de classe en Java
- Le programme principal en Java
- Énumération en Java

# Métaclasse

- La relation qui existe entre objet et classe est une relation d'instanciation
- De même, on peut considérer une classe comme une instance de classe de plus « haut niveau », dite *métaclasse*
- Cette notion permet d'introduire des *méthodes de classe* et des *attributs de classe*, i.e. des membres associés à la classe et non pas à une instance particulière
- Une *méthode de classe* peut être invoquée sans instance particulière
- Un *attribut de classe* est associé à la classe elle-même et non pas à une instance particulière

# Exemple

## Membres de classe

Cercle2DWithCpt	
~	<u>nblInstances=0</u>
-	centre:Point2D
-	rayon:Réel
<hr/>	
+	Cercle2DWithCpt(in centre, in rayon)
+	<u>getNbInstances()</u>
#	finalize()



# Chapitre IV : Objet et classe

## Section 13 : Métaclasse

- Métaclasse et membres de classe
- **Membres de classe en Java**
- Le programme principal en Java
- Énumération en Java

# Membres de classe en Java

- Un membre de classe se déclare avec le mot-clé `static`
- L'accès à un membre de classe se fait par l'intermédiaire de la classe
- Pour un attribut de classe, le système alloue un espace mémoire pour un attribut par classe (et non pas un attribut par instance)
- Un attribut de classe est souvent utilisé pour définir une constante (`static final`)

```
public static final double E = 2.718281828459045d;  
public static final double PI = 3.141592653589793d;
```

- L'initialisation d'un attribut de classe peut se faire directement ou en utilisant un *bloc d'initialisation statique*
- Un *bloc d'initialisation statique* est un bloc de code Java classique commençant par le mot-clé `static` et placé dans le corps de la classe
- Une méthode de classe ne peut pas accéder aux attributs d'instance

# Exemple

## Compter les instances de cercle 1/2

```
class Cercle2DWithCpt {  
    /** Le nombre d'instances de Cercle2DWithCpt */  
    static int nbInstances = 0;  
  
    /** Le centre du cercle. */  
    private Point2D centre;  
  
    /** Le rayon du cercle */  
    private double rayon;  
  
    /**  
     * Initialise un cercle avec un centre et un rayon.  
     * @param centre Le centre.  
     * @param rayon Le rayon.  
     */  
    public Cercle2DWithCpt(Point2D centre, double rayon) {  
        this.centre = centre;  
        this.rayon = rayon;  
        ++nbInstances;  
    }  
}
```

La classe Cercle2DWithCpt (part. 1)

# Exemple

## Compter les instances de cercle 2/2

```
/**
 * Retourne le nombre d'instances de la classe.
 * @return le nombre d'instances.
 */
public static int getNbInstances() {
    return nbInstances;
}

/**
 * Decrementation du nombre d'instances quand l'objet est detruit.
 */
@Override
protected void finalize() throws Throwable {
    —nbInstances;
    super.finalize();
}
}
```

La classe Cercle2DWithCpt (part. 2)

# Exemple

## Invoquer une méthode de classe

```
assert Cercle2DWithCpt.getNbInstances() == 0 :
    Cercle2DWithCpt.getNbInstances();

// Creation des cercles
Cercle2DWithCpt c1 = new Cercle2DWithCpt(new Point2D(2.0, 4.0),
    5.0);
Cercle2DWithCpt c2 = new Cercle2DWithCpt(new Point2D(1.0, 2.0),
    4.0);
Cercle2DWithCpt c3 = new Cercle2DWithCpt(new Point2D(3.0, 4.0),
    2.0);

assert Cercle2DWithCpt.getNbInstances() == 3 :
    Cercle2DWithCpt.getNbInstances();

// Simple liaison
Cercle2DWithCpt unAutreC3 = c3;

assert Cercle2DWithCpt.getNbInstances() == 3 :
    Cercle2DWithCpt.getNbInstances();

// Suppression d'un instance
c1 = null; System.gc(); Thread.sleep(1000);

assert Cercle2DWithCpt.getNbInstances() == 2 :
    Cercle2DWithCpt.getNbInstances();
```

## Invoquer une méthode de classe

# Chapitre IV : Objet et classe

## Section 13 : Métaclasse

- Métaclasse et membres de classe
- Membres de classe en Java
- Le programme principal en Java
- Énumération en Java

# Rôle du programme principal

- Un système OO en cours d'exécution est une collection d'objets qui interagissent
- Le lancement du programme doit donc permettre d'instancier ces objets
- En général, le programme principal se limite à créer un *objet application* qui se charge d'instancier les autres objets

# Le programme principal en Java

## La méthode `main`

- Le point d'entrée d'une application Java est une méthode de classe nommée `main`
- Lors de l'exécution, l'interpréteur Java est invoqué avec le nom d'une classe qui doit implémenter une méthode `main`
- La déclaration de la méthode `main` est : `public static void main(String[] args)`
- Le paramètre de `main` est un tableau de chaînes de caractères contenant les *arguments de ligne de commande* passés lors de l'appel du programme
- On limite en général le code se trouvant dans le `main` au strict minimum : création d'un objet application et invocation d'une méthode



# Exemple

## Le programme principal en Java (avec une énumération)

```
/**
 * Représente l'application.
 *
 * @version   jan. 2015
 * @author    Stéphane Lopes
 */
enum ApplicationVide {
    ENVIRONNEMENT;

    /**
     * Méthode principale du programme.
     * @param args les arguments de ligne de commande
     */
    public void run(String[] args) {
        // ...
    }

    /**
     * Point d'entrée du programme.
     * @param args les arguments de ligne de commande
     */
    public static void main(String[] args) {
        ENVIRONNEMENT.run(args);
    }
}
```

## Le programme principal en Java

# Chapitre IV : Objet et classe

## Section 13 : Métaclasse

- Métaclasse et membres de classe
- Membres de classe en Java
- Le programme principal en Java
- Énumération en Java

# Complément sur le type énuméré

- Un type énuméré en Java est en fait une classe dont les instances sont connues lors de la compilation
- Les constantes d'un type énuméré peuvent être utilisées partout où un objet peut l'être
- Un type énuméré peut donc contenir des méthodes et des attributs
- De plus, le compilateur ajoute automatiquement certaines méthodes
  - `values()` retourne un tableau contenant les constantes dans l'ordre de leur déclaration
  - un type énuméré hérite implicitement de la classe `Enum`

# Chapitre IV : Objet et classe

## Section 14 : Généricité

- Généricité
- Généricité en Java

# Chapitre IV : Objet et classe

## Section 14 : Généricité

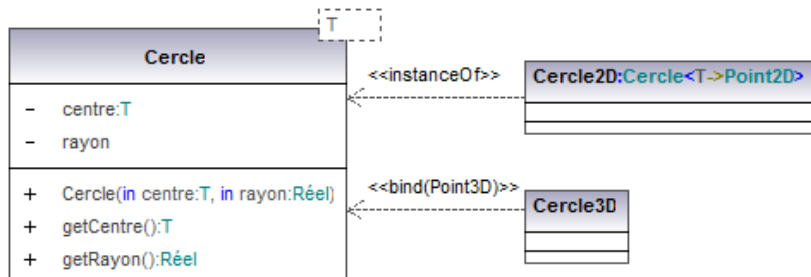
- Généricité
- Généricité en Java

# Généricité

- La *généricité* permet de paramétrer une classe par un ou plusieurs paramètres formels (généralement des types)
- La généricité permet de définir une famille de classes, chaque classe étant instanciée lors du passage des paramètres effectifs
- Une classe paramétrée (ou générique) est donc une métaclasse particulière
- Il peut être souhaitable de limiter les paramètres possibles : la généricité est alors *contrainte*
- Utiliser un type paramétré améliore la vérification de type lors de la compilation
- Cette notion est orthogonale au paradigme OO : on parle de *programmation générique*

# Exemple

## Un cercle générique



# Chapitre IV : Objet et classe

## Section 14 : Généricité

- Généricité
- Généricité en Java



# Généricité en Java

- Les *paramètres formels de type* sont placés entre « < » et « > »
- Un paramètre effectif est obligatoirement une classe (pas un type primitif)
- Le mécanisme implémentant la généricité en Java se nomme *Type erasure*
- Ce mécanisme supprime toute trace de la généricité dans le bytecode  
⇒ il n'existe pas d'information concernant la généricité à l'exécution

# Classe générique en Java

- Les paramètres formels de type sont placés entre « < » et « > » juste après le nom de la classe

```
class Cercle<T> {  
    // ...  
}
```

- Le paramètre de type peut ensuite être utilisé comme tout autre type dans la définition de la classe
- La déclaration d'une variable de ce type nécessite de passer le type effectif à la classe paramétrée

```
Cercle<Point2D> c = // ...
```

- la création d'une instance nécessite également le passage du type effectif

```
Cercle<Point2D> c = new Cercle<Point2D>(/ * ... */);
```

# Exemple

## Définition de la classe générique Cercle

```
class Cercle<T> {  
    /** Le centre du cercle. */  
    private T centre;  
  
    /** Le rayon du cercle */  
    private double rayon;  
  
    /**  
     * Initialise un cercle avec un centre et un rayon.  
     * @param centre Le centre.  
     * @param rayon Le rayon.  
     */  
    public Cercle(T centre, double rayon) {  
        this.centre = centre;  
        this.rayon = rayon;  
    }  
  
    public T getCentre() {  
        return centre;  
    }  
  
    public double getRayon() {  
        return rayon;  
    }  
}
```

Une classe Cercle générique

# Exemple

## Utilisation de la classe générique Cercle

```
// En 2 dimensions
Point2D p1 = new Point2D(1.0, 2.0);
Cercle<Point2D> c1 = new Cercle<Point2D>(p1, 3.0);

// Invocation d'une methode de Point2D
double xCentre = c1.getCentre().getAbscisse();

// En 3 dimensions
Point3D p2 = new Point3D(3.0, 4.0, 5.0);
Cercle<Point3D> c2 = new Cercle<Point3D>(p2, 3.0);

// Invocation d'une methode de Point3D
double hCentre = c2.getCentre().getHauteur();
```

## Utilisation de la classe générique Cercle

# Méthode générique en Java

- Il est possible de déclarer des méthodes génériques
- Le paramètre de type formel est alors précisé avant la déclaration

```
public static <T> T max(T o1, T o2) // ...
```

- La portée de ce paramètre est alors restreinte à la méthode
- L'invocation de la méthode peut préciser le type effectif ou se baser sur l'*inférence de type*

```
// Type effectif explicite  
Integer i = uneClasse.<Integer>max(i1, i2);  
  
// Type effectif déterminé par inférence de type  
Integer i = uneClasse.max(i1, i2);
```

# Généricité contrainte en Java

- Il est possible d'imposer que le paramètre de type formel soit un sous-type d'un autre type avec le mot-clé `extends`

```
public static <T extends Number> T max(T o1, T o2) // ...
```

- Le mot-clé `super` permet d'imposer que le paramètre de type formel soit un super-type d'un autre type (exemple : `<T super Number>`)
- Il peut être nécessaire d'utiliser le caractère *joker* ? si le type effectif n'est pas connu

```
// Une boîte qui peut contenir des nombres  
Boite<? extends Number> b = //...
```

```
// Un boîte qui peut contenir n'importe quoi  
Boite<?> b = //...
```

# Chapitre V : Héritage en Java

15 Héritage

16 Polymorphisme

17 Classe abstraite

18 Interface

# Chapitre V : Héritage en Java

## Section 15 : Héritage



# Héritage en Java

- On spécifie qu'une classe est une sous-classe d'une autre en utilisant `extends` dans la déclaration `class NomClasse extends NomSuperClasse //...`
  - Une classe ne peut avoir qu'une seule super-classe
  - Si `extends` n'est pas précisé, la classe hérite de la classe `Object`
- ⇒ une classe Java a une et une seule super-classe
- Une classe déclarée `final` ne peut plus être spécialisée

# Héritage et membres

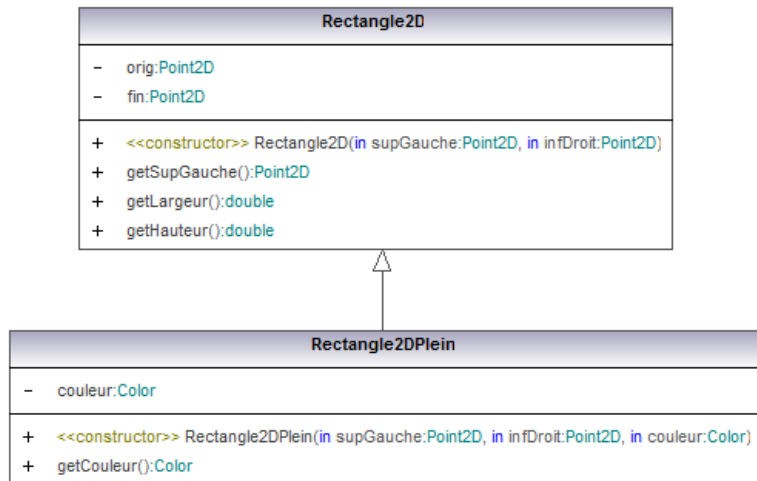
- Une classe  $C$  hérite de sa super-classe  $S$  les attributs et méthodes qu'elle possède
  - tous les attributs de  $S$  font parti de l'état des instances de  $C$
  - les attributs privées de  $S$  ne sont pas accessibles dans  $C$  (mais font parti des instances de  $C$ )
  - les méthodes non privées de  $S$  sont accessibles dans  $C$
  - les méthodes publiques de  $S$  font parti de l'interface publique de  $C$
  - les constructeurs de  $S$  sont utilisables dans les constructeurs de  $C$  mais ne font pas parti de l'interface publique de  $C$

# Masquage de membres

- Une classe peut masquer un membre de sa super-classe si elle possède un membre de même nom (ou de même signature)
- Le mot clé `super` permet d'accéder aux membres masqués d'une super-classe

# Exemple

## Rectangle et rectangle plein



# Exemple

## Définition de la classe Rectangle2DPlein en Java

```
class Rectangle2DPlein extends Rectangle2D {  
    /** La couleur de remplissage */  
    private Color couleur;  
  
    /**  
     * Initialise le rectangle plein.  
     * @param supGauche Le coin supérieur gauche.  
     * @param infDroit Le coin inférieur droit.  
     * @param couleur La couleur de remplissage.  
     */  
    public Rectangle2DPlein(Point2D supGauche,  
        Point2D infDroit,  
        Color couleur) {  
        super(supGauche, infDroit);  
        assert couleur != null;  
        this.couleur = couleur;  
    }  
  
    /**  
     * Renvoie la couleur.  
     * @return la couleur.  
     */  
    public Color getCouleur() { return couleur; }  
}
```

la classe Rectangle2DPlein en Java

# Exemple

## Utilisation de la classe Rectangle2DPlein

```
// Déclaration et création d'un rectangle rouge
Rectangle2DPlein rp1 = new Rectangle2DPlein(new Point2D(1.0, 2.0),
                                             new Point2D(3.0, 0.0),
                                             Color.RED);

assert rp1.getCouleur() == Color.RED;

// Déclaration d'un rectangle et
// liaison avec un rectangle plein
Rectangle2D r1 = new Rectangle2DPlein(new Point2D(1.0, 2.0),
                                       new Point2D(2.0, 1.0),
                                       Color.YELLOW);

assert r1.getLargeur() == 1;
```

## Utilisation de la classe Rectangle2DPlein

# Exercice

## L'héritage

Deux nouveaux types de robot sont créés :

- les transporteurs qui peuvent ramasser un objet, le transporter et le déposer,
- les destructeurs qui peuvent détruire ce qui se trouve sur la case devant eux.

① Modéliser les robots

② Soit le terrain suivant

T S	M	
D E	M	
	M	

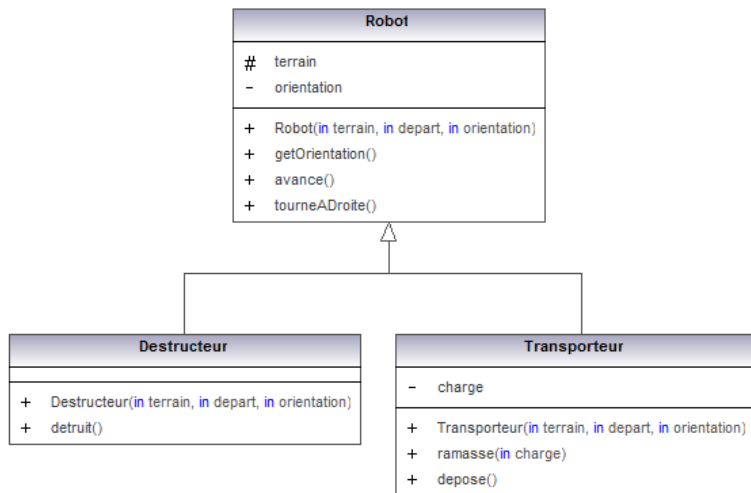
(M = mur, T = transporteur,

D = destructeur, S = sud, E = est). Écrire les instructions permettant de créer ce terrain puis de déplacer l'objet se trouvant en (0, 0) en (2, 2).

③ Donner l'implémentation des deux classes Transporteur et Destructeur.

# Exercice (solutions)

## Modélisation





# Exercice (solutions)

## Déplacement des robots

```
// Création du terrain
Position [] murs = { new Position(1, 0), new Position(1, 1),
                     new Position(1, 2) };
Terrain terrain = new Terrain(3, 3, murs);

// Création des robots
Transporteur transp = new Transporteur(terrain, new Position(0, 0),
                                       Direction.SUD);
Destructeur destr = new Destructeur(terrain, new Position(0, 1),
                                    Direction.EST);

// Actions des robots
destr.detruit();
destr.tourneADroite();
destr.avance();
transp.ramasse(new Object());
transp.avance();
transp.tourneADroite(); transp.tourneADroite(); transp.tourneADroite();
transp.avance();
transp.avance();
transp.tourneADroite();
transp.avance();
transp.depose();
```

## Déplacement des robots

# Exercice (solutions)

## Classe Transporteur 1/2

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de transporter un objet.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Transporteur extends Robot {
    /** L'objet transporté. */
    private Object charge;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Transporteur(Terrain terrain, Position depart,
                       Direction orientation) {
        super(terrain, depart, orientation);
    }
}
```

## Classe Transporteur (part. 1)

# Exercice (solutions)

## Classe Transporteur 2/2

```
/**
 * Prends un objet sur le terrain.
 * @param charge objet à transporter.
 */
public void ramasse(Object charge) {
    this.charge = charge;
}

/**
 * Dépose l'objet transporté.
 * @return l'objet transporté.
 */
public Object depose() {
    Object tmp = charge;
    charge = null;
    return tmp;
}
```

## Classe Transporteur (part. 2)

# Exercice (solutions)

## Classe Destructeur

```

/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de détruire un obstacle
 * se trouvant devant eux.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Destructeur extends Robot {
    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Destructeur(Terrain terrain, Position depart,
                       Direction orientation) {
        super(terrain, depart, orientation);
    }

    /**
     * Supprime le contenu de la case se trouvant devant le robot
     */
    public void detruit() {
        terrain.detruitCaseDevant(this);
    }
}

```

## Classe Destructeur

# Chapitre V : Héritage en Java

## Section 16 : Polymorphisme

- Polymorphisme en Java
- La classe Object

# Chapitre V : Héritage en Java

## Section 16 : Polymorphisme

- Polymorphisme en Java
- La classe `Object`

# Redéfinition de méthode

- Une sous-classe peut *redéfinir* (*override*) une ou plusieurs méthodes de sa super-classe
- La *redéfinition* (*overriding*) consiste à définir dans une sous-classe, une méthode ayant même signature et même type de retour qu'une méthode de la super-classe
  - la méthode de la super-classe est alors masquée
  - il est toujours possible d'appeler la méthode redéfinie en utilisant le mot-clé `super`

# Redéfinition et polymorphisme

- Le polymorphisme est le mécanisme permettant de sélectionner la méthode redéfinie appropriée
- La redéfinition ne permet plus au compilateur de sélectionner la méthode adéquat
- C'est le type de l'objet (et non pas de la référence) qui permettra de déterminer la méthode à invoquer et ce type ne peut être connu qu'au moment de l'exécution

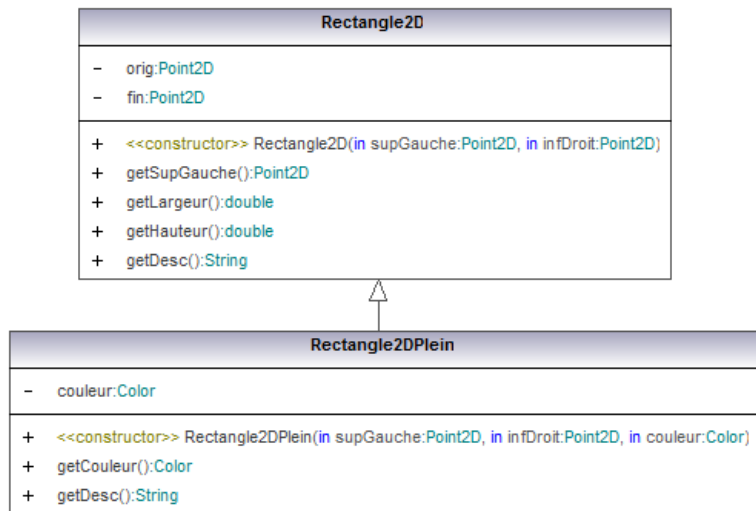


# Compléments sur la redéfinition de méthode

- La déclaration de la méthode redéfinie est toujours précédée de l'annotation `@Override`
- Le contrôle d'accès peut être relaxé lors de la redéfinition
- Une méthode déclarée `final` ne peut pas être redéfinie
- Une méthode de classe ne peut pas être redéfinie

# Exemple

Une description pour le rectangle et le rectangle plein



# Exemple

## La classe Rectangle2D 1/2

```
class Rectangle2D implements Cloneable {  
    /** Coordonnées du coin supérieur gauche */  
    private Point2D orig;  
  
    /** Coordonnées du coin inférieur droit */  
    private Point2D fin;  
  
    /**  
     * Initialise le rectangle.  
     * @param supGauche Le coin supérieur gauche.  
     * @param infDroit Le coin inférieur droit.  
     */  
    public Rectangle2D(Point2D supGauche, Point2D infDroit) {  
        assert supGauche.getAbscisse() <= infDroit.getAbscisse() &&  
            supGauche.getOrdonnee() >= infDroit.getOrdonnee();  
        orig = supGauche;  
        fin = infDroit;  
    }  
  
    public Point2D getSupGauche() { return orig; }  
  
    public double getLargeur() {  
        return fin.getAbscisse() - orig.getAbscisse();  
    }  
  
    public double getHauteur() {  
        return orig.getOrdonnee() - fin.getOrdonnee();  
    }  
}
```

# Exemple

## La classe Rectangle2D 1/2

```
/**
 * Retourne une description du rectangle.
 * @return la description.
 */
public String getDesc() {
    StringBuilder str = new StringBuilder();
    str.append("O = ");
    str.append(orig.getAbscisse());
    str.append(", ");
    str.append(orig.getOrdonnee());
    str.append(") L = "); str.append(getLargeur());
    str.append(" H = "); str.append(getHauteur());
    return str.toString();
}
```

## La classe Rectangle2D (part. 2)

# Exemple

## La classe Rectangle2DPlein 1/2

```
class Rectangle2DPlein extends Rectangle2D {  
    /** La couleur de remplissage */  
    private Color couleur;  
  
    /**  
     * Initialise le rectangle plein.  
     * @param supGauche Le coin supérieur gauche.  
     * @param infDroit Le coin inférieur droit.  
     * @param couleur La couleur de remplissage.  
     */  
    public Rectangle2DPlein(Point2D supGauche,  
        Point2D infDroit,  
        Color couleur) {  
        super(supGauche, infDroit);  
        assert couleur != null;  
        this.couleur = couleur;  
    }  
  
    /**  
     * Renvoie la couleur.  
     * @return la couleur.  
     */  
    public Color getCouleur() { return couleur; }
```

La classe Rectangle2DPlein (part. 1)

# Exemple

## La classe Rectangle2DPlein 2/2

```
/**
 * Retourne une description du rectangle plein.
 * @return la description.
 */
@Override
public String getDesc() {
    StringBuilder str = new StringBuilder(super.getDesc());
    str.append(", couleur : (");
    str.append(couleur.getRed());
    str.append(", ");
    str.append(couleur.getGreen());
    str.append(", ");
    str.append(couleur.getBlue());
    str.append(")");
    return str.toString();
}
```

La classe Rectangle2DPlein (part. 2)

# Exemple

## Utilisation du polymorphisme

```
// Création d'un tableau de références sur des Rectangle2D
final int NB_RECTANGLES = 2;
Rectangle2D[] figures = new Rectangle2D[NB_RECTANGLES];

// Un rectangle
figures[0] = new Rectangle2D(new Point2D(0.0, 5.0),
    new Point2D(2.0, 2.0));

// Un rectangle plein
figures[1] = new Rectangle2DPlein(new Point2D(1.0, 3.0),
    new Point2D(3.0, 2.0),
    Color.BLUE);

// getDesc() de Rectangle2D
assert figures[0].getDesc().equals("O = (0.0, 5.0) L = 2.0 H = 3.0");

// getDesc() de Carre2D
assert figures[1].getDesc().equals("O = (1.0, 3.0) L = 2.0 H = 1.0, couleur : (0, 0, 255)");
```

## Utilisation du polymorphisme en Java

# Exercices

## Le polymorphisme et la redéfinition

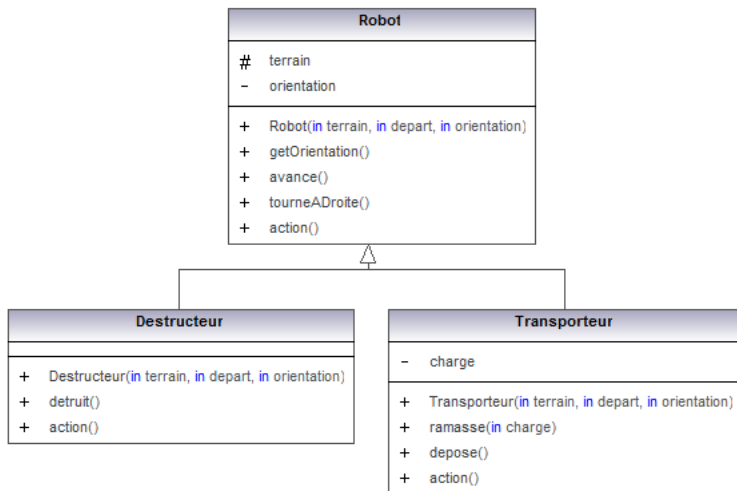
Une amélioration importante a été apportée aux robots : ils sont maintenant programmables. Chaque type de robot possède son propre comportement. Quand ils en reçoivent l'ordre, ils exécutent l'action programmée (un ensemble d'instructions élémentaires). Plusieurs robots de types différents se trouvent sur le terrain. On veut pouvoir déclencher l'exécution du programme de l'ensemble des robots.

- 1 Modéliser cet énoncé
- 2 Implémenter les changements dans les classes Robot, Transporteur et Destructeur
- 3 Écrire un programme déclenchant l'exécution de l'action pour l'ensemble des robots



# Exercice (solutions)

## Modélisation



# Exercice (solutions)

## La classe Robot

```
public class Robot {  
    /** Terrain sur lequel évolue le robot. */  
    protected Terrain terrain;  
  
    /** Orientation du robot. */  
    private Direction orientation;  
  
    /**  
     * Initialise le robot.  
     * @param terrain terrain sur lequel le robot va évoluer  
     * @param direction orientation du robot  
     */  
    public Robot(Terrain terrain, Position depart, Direction orientation) {  
        assert terrain != null;  
        this.terrain = terrain;  
        terrain.ajouterRobot(this, depart);  
        this.orientation = orientation;  
    }  
  
    public Direction getOrientation() { return orientation; }  
    public boolean avance() { return terrain.avancerRobot(this); }  
    public void tourneADroite() { orientation = orientation.next(); }  
  
    public void action() {  
        for (int i = 0; i < 4; ++i) {  
            avance(); avance();  
            tourneADroite();  
        }  
    }  
}
```

# Exercice (solutions)

## La classe Transporteur

```
class Transporteur extends Robot {  
    /** L'objet transporté. */  
    private Object charge;  
  
    public Transporteur(Terrain terrain, Position depart,  
        Direction orientation) {  
        super(terrain, depart, orientation);  
    }  
  
    public void ramasse(Object charge) { this.charge = charge; }  
  
    public Object depose() {  
        Object tmp = charge;  
        charge = null;  
        return tmp;  
    }  
  
    public void action() {  
        super.action();  
        ramasse(new Object());  
        avance();  
        depose();  
    }  
}
```

## La classe Transporteur

# Exercice (solutions)

## La classe Destructeur 1/2

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de détruire un obstacle
 * se trouvant devant eux.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Destructeur extends Robot {
    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Destructeur(Terrain terrain, Position depart,
                      Direction orientation) {
        super(terrain, depart, orientation);
    }
}
```

## La classe Destructeur (part. 1)

# Exercice (solutions)

## La classe Destructeur 2/2

```
/**
 * Supprime le contenu de la case se trouvant devant le robot
 */
public void detruit() {
    terrain.detruitCaseDevant(this);
}

/**
 * Décrit la programmation du robot.
 */
public void action() {
    for (int i = 0; i < 4; ++i) {
        detruit(); avance();
    }
}
```

## La classe Destructeur (part. 2)

# Exercice (solutions)

## La classe Application

```
// Création du terrain
Terrain terrain = new Terrain(4, 5, null);

// Création des robots
final int NB_ROBOTS = 3;
Robot[] robots = new Robot[NB_ROBOTS];
robots[0] = new Robot(terrain, new Position(0, 0),
                      Direction.EST);
robots[1] = new Transporteur(terrain, new Position(1, 1),
                             Direction.EST);
robots[2] = new Destructeur(terrain, new Position(3, 0),
                            Direction.SUD);

// Exécution des actions des robots
for (int i = 0; i < NB_ROBOTS; ++i) {
    robots[i].action();
}
```

## La classe Application

# Chapitre V : Héritage en Java

## Section 16 : Polymorphisme

- Polymorphisme en Java
- La classe `Object`

# La classe Object

- La classe Object définit et implémente le comportement dont chaque classe Java a besoin
- C'est la plus générale des classes Java
- Chaque classe Java hérite directement ou indirectement de Object (tout objet y compris les tableaux implémente les méthodes de Object)



# Les méthodes de la classe Object

- Certaines méthodes de Object peuvent être redéfinies pour s'adapter à la sous-classe
  - `protected Object clone()` permet de dupliquer un objet
  - `boolean equals(Object obj)` permet de tester l'égalité de deux objets et `int hashCode()` de renvoyer une valeur de hashage
  - `protected void finalize()` représente le destructeur d'un objet
  - `String toString()` retourne une chaîne représentant l'objet
- Autres méthodes
  - `Class getClass()` retourne un objet de type Class représentant la classe de l'objet
  - quelques méthodes pour les *threads*

# Exemple

## Redéfinition de la méthode toString() de Rectangle2D

```
/**
 * Retourne une chaîne représentant l'objet.
 * @return la chaîne.
 */
@Override
public String toString() {
    StringBuilder str = new StringBuilder();
    str.append("O = "); str.append(orig.toString());
    str.append(" L = "); str.append(getLargeur());
    str.append(" H = "); str.append(getHauteur());
    return str.toString();
}
```

## Méthode toString() de Rectangle2D

# Exemple

## Redéfinition de la méthode toString() de Rectangle2DPlein

```
/**
 * Retourner une chaîne représentant l'objet.
 * @return la chaîne.
 */
@Override
public String toString() {
    StringBuilder str = new StringBuilder(super.toString());
    str.append(" couleur = "); str.append(couleur.toString());
    return str.toString();
}
```

## Méthode toString() de Rectangle2DPlein

```
// Test de toString
assert figures[0].toString().equals("O = (0.0, 5.0) [5.0, 0.0] L = 2.0 H = 3.0");
assert figures[1].toString().equals("O = (1.0, 3.0) [3.1622776601683795, 0.3217505543966422] L = 2.0 H = 1.0 couleur = java.awt.Color[r=0,g=0,b=255]");
```

## Appel de toString()

# Copie d'objet

## Le mécanisme de clonage

- Le *clonage* d'objet est destinée à créer une *copie profonde* (*deep copy*) d'un objet
- Le clonage est réalisé en redéfinissant en public la méthode protégée `clone` de `Object`
- La méthode `clone` de `Object` doit être appelée pour réserver la mémoire et effectuer une *copie superficielle* (*shallow copy*)
- L'interface `Cloneable`, qui ne comporte aucune méthodes, doit aussi être implémentée
- Les conditions suivantes doivent être vérifiées :
  - la copie et l'objet ne sont pas identiques (`x.clone() != x`),
  - la copie et l'objet sont de même classe (`x.clone().getClass() == x.getClass()`),
  - la copie et l'objet sont égaux (`x.clone().equals(x)`).
- Rappel : l'affectation d'un objet a une variable ne crée pas de copie de l'objet mais juste une nouvelle référence vers l'objet

# Exemple

## Clonage d'un rectangle

```
/**
 * Retourne une copie "profonde" de l'objet.
 * @return la copie.
 */
@Override
public Object clone() throws CloneNotSupportedException {
    Rectangle2D r = (Rectangle2D) super.clone();
    r.orig = (Point2D) orig.clone();
    r.fin = (Point2D) fin.clone();
    return r;
}
```

## Redéfinition de clone dans Rectangle2D

```
// Test du clonage d'objets
Rectangle2D copie = (Rectangle2D) figures[0].clone();
assert copie != figures[0]; // Pas identiques
assert copie.getClass() == figures[0].getClass(); // Même classe
assert copie.equals(figures[0]); // Egaux
```

## Utilisation de clone

# Egalité d'objets

## La méthode equals

- La méthode boolean `equals(Object o)` teste l'égalité de deux objets
- La méthode `equals` de la classe `Object` se content de tester l'égalité des références des objets, i.e. l'identité
- Il est donc en général nécessaire de redéfinir `equals` pour le test d'égalité
- Rappel : l'opérateur `==` teste l'identité de ses opérandes, i.e. l'égalité des références

# Egalité d'objets

## Contraintes de equals

- equals implémente une relation d'équivalence pour des références d'objet non nulles
  - `x.equals(x) == true`
  - `x.equals(y) == true` si et seulement si `y.equals(x) == true`
  - si `x.equals(y) == true` et `y.equals(z) == true` alors `x.equals(z) == true`
  - `x.equals(null) == false`
- Toute classe qui redéfinit equals doit également redéfinir hashCode()
  - si deux objets sont égaux au sens de equals alors hashCode doit produire le même résultat pour les deux objets

# Exemple

## Egalité de rectangles

```
/**
 * Teste l'égalité de deux rectangles.
 * @param obj le rectangle à comparer.
 * @return true si les objets sont égaux.
 */
@Override
public boolean equals(Object obj) {
    if (obj instanceof Rectangle2D) {
        Rectangle2D r = (Rectangle2D) obj;
        return orig.equals(r.orig) && fin.equals(r.fin);
    }
    return false;
}

/**
 * Retourne une valeur de hashage pour l'objet.
 * @return la valeur de hashage.
 */
@Override
public int hashCode() {
    return orig.hashCode() ^ fin.hashCode();
}
```

Redéfinition de equals et hashCode() dans Rectangle2D



# Exemple

## Contraintes de equals

```
// Test de l'égalité
Rectangle2D r1 = new Rectangle2D(new Point2D(0.0, 5.0),
    new Point2D(2.0, 2.0));
Rectangle2D r2 = new Rectangle2D(new Point2D(0.0, 5.0),
    new Point2D(2.0, 2.0));
Rectangle2D r3 = new Rectangle2D(new Point2D(0.0, 5.0),
    new Point2D(2.0, 2.0));
assert r1.equals(r1); // Réflexivité
assert r1.equals(r2) && r2.equals(r1); // Symétrie
assert r1.equals(r2) && r2.equals(r3) &&
    !r1.equals(r3) == false; // Transitivité
assert r1.equals(null) == false;
assert r1.hashCode() == r2.hashCode();
```

## Contraintes de equals

# Chapitre V : Héritage en Java

## Section 17 : Classe abstraite

- Classe abstraite en Java
- La classe Number

# Chapitre V : Héritage en Java

## Section 17 : Classe abstraite

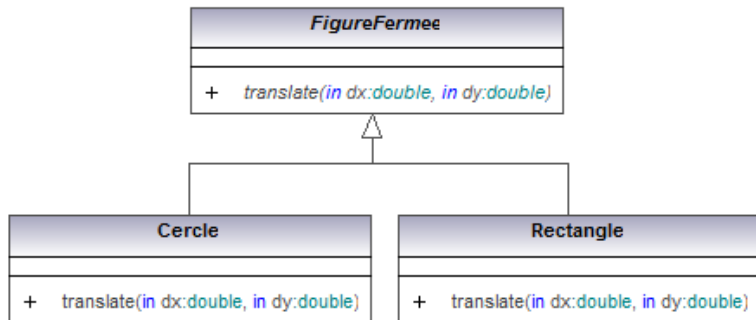
- Classe abstraite en Java
- La classe `Number`

# Classe abstraite en Java

- Une classe est spécifiée abstraite en ajoutant le mot-clé `abstract` dans sa déclaration
- L'instanciation d'une telle classe est alors refusée par le compilateur
- Une classe abstraite contient généralement des *méthodes abstraites*, i.e. qui ne possèdent pas d'implémentation
  - une classe abstraite peut cependant ne pas avoir de méthodes abstraites
- Une méthode est déclarée abstraite en utilisant le mot-clé `abstract` lors de sa déclaration
- Toute sous-classe non abstraite d'une classe abstraite doit redéfinir les méthodes abstraites de cette classe
- Une classe possédant des méthodes abstraites est obligatoirement abstraite

# Exemple

## La hiérarchie d'héritage des figures



# Exemple

## La classe abstraite FigureFermee2D

```
/**
 * Une figure fermée.
 *
 * @version   oct. 2008
 * @author    Stéphane Lopes
 */
abstract class FigureFermee2D {
    /**
     * Translate la figure.
     * @param dx déplacement en abscisse.
     * @param dy déplacement en ordonnées.
     */
    public abstract void translate(double dx, double dy);
}
```

## La classe abstraite FigureFermee2D

# Exemple

## Redéfinition de la translation

```
/**
 * Translate le rectangle.
 * @param dx déplacement en abscisse.
 * @param dy déplacement en ordonnées.
 */
@Override
public void translate(double dx, double dy) {
    orig.translate(dx, dy);
    fin.translate(dx, dy);
}
```

### Translation dans la classe Rectangle2D

```
/**
 * Translate le cercle.
 * @param dx déplacement en abscisse.
 * @param dy déplacement en ordonnées.
 */
@Override
public void translate(double dx, double dy) {
    centre.translate(dx, dy);
}
```

### Translation dans la classe Cercle2D

# Exemple

## Utilisation de la classe abstraite FigureFermee2D

```
// Création du tableau de références
final int NB_FIGURES = 4;
FigureFermee2D[] figures = new FigureFermee2D[NB_FIGURES];

// Création des formes
figures[0] = new Rectangle2D(new Point2D(0.0, 5.0),
    new Point2D(2.0, 2.0));
figures[1] = new Cercle2D(new Point2D(1.0, 2.0), 3.0);
figures[2] = new Rectangle2D(new Point2D(5.0, 5.0),
    new Point2D(7.0, 3.0));
figures[3] = new Cercle2D(new Point2D(4.0, 5.0), 2.0);

// Réalise une translation de la figure
for (int i = 0; i < figures.length; ++i) {
    figures[i].translate(1.0, 2.0);
}
```

## Utilisation de la classe abstraite FigureFermee2D



# Exercices

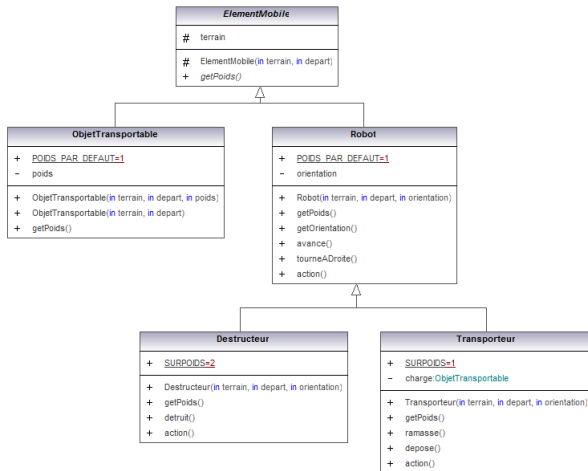
## Les classes abstraites

On souhaite maintenant ajouter sur le terrain un type de case sensible à la charge (pont par exemple). Chaque élément mobile (robot ou objet transportable) devra donc posséder un poids. Le poids d'une instance de robot sera toujours de 1 unité. Une instance de destructeur aura un poids de 2 unités de plus que le robot. Une instance de transporteur aura un poids de 1 unités de plus que le robot auquel il faudra ajouter le poids de l'objet transporté. Le poids par défaut des objets transportables est de 1 unité mais chaque objet pourra avoir un poids différent précisé lors de sa création. On souhaite pouvoir connaître le poids de tout élément se trouvant sur le terrain.

- 1 Modéliser la prise en compte du poids au niveau des éléments mobiles
- 2 Implémenter les classes correspondantes

# Exercice (solutions)

## Modélisation



# Exercice (solutions)

## La classe ElementMobile

```
public abstract class ElementMobile {  
    /** Terrain sur lequel se trouve l'élément. */  
    protected Terrain terrain;  
  
    /**  
     * Place l'élément sur un terrain à une position donnée.  
     * @param terrain terrain sur lequel l'élément va évoluer  
     * @param depart position de départ sur le terrain  
     */  
    protected ElementMobile(Terrain terrain, Position depart) {  
        assert terrain != null && depart != null;  
        this.terrain = terrain;  
        terrain.ajouter(this, depart);  
    }  
  
    /**  
     * Retourne le poids de l'élément.  
     * @return le poids de l'élément.  
     */  
    public abstract int getPoids();  
}
```

## La classe ElementMobile

# Exercice (solutions)

## La classe `ObjetTransportable`

```
public class ObjetTransportable extends ElementMobile {  
    /** Le poids par défaut des objets */  
    public static final int POIDS_PAR_DEFAUT = 1;  
  
    /** Le poids de l'objet */  
    private int poids;  
  
    /**  
     * Initialise l'objet.  
     * @param terrain terrain sur lequel l'objet se trouve  
     * @param depart position de départ sur le terrain  
     * @param poids le poids de l'objet.  
     */  
    public ObjetTransportable(Terrain terrain, Position depart, int poids) {  
        super(terrain, depart);  
        this.poids = poids;  
    }  
  
    /**  
     * Initialise un objet de poids 1.  
     * @param terrain terrain sur lequel l'objet se trouve  
     * @param depart position de départ sur le terrain  
     */  
    public ObjetTransportable(Terrain terrain, Position depart) {  
        this(terrain, depart, POIDS_PAR_DEFAUT);  
    }  
  
    /**  
     * Retourne le poids de l'élément.
```

# Exercice (solutions)

## La classe Robot 1/2

```
/**
 * Cette classe représente un robot pouvant évoluer sur un terrain.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
public class Robot extends ElementMobile {
    /** Le poids par défaut des robots */
    public static final int POIDS_PAR_DEFAUT = 1;

    /** Orientation du robot. */
    private Direction orientation;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer
     * @param depart position de départ sur le terrain
     * @param direction orientation du robot
     */
    public Robot(Terrain terrain, Position depart, Direction orientation) {
        super(terrain, depart);
        this.orientation = orientation;
    }
}
```

## La classe Robot (part. 1)

# Exercice (solutions)

## La classe Robot 2/2

```
/**  
 * Retourne le poids du robot.  
 * @return le poids de l'élément.  
 */  
@Override  
public int getPoids() {  
    return POIDS_PAR_DEFAULT;  
}
```

## La classe Robot (part. 2)

# Exercice (solutions)

## La classe Destructeur

```
class Destructeur extends Robot {  
    /** Surpoids des robots destructeurs */  
    public static final int SURPOIDS = 2;  
  
    /**  
     * Initialise le robot.  
     * @param terrain terrain sur lequel le robot va évoluer.  
     * @param depart position de départ du robot.  
     * @param direction orientation du robot.  
     */  
    public Destructeur(Terrain terrain, Position depart, Direction orientation) {  
        super(terrain, depart, orientation);  
    }  
  
    /**  
     * Retourne le poids du robot.  
     * @return le poids de l'élément.  
     */  
    @Override  
    public int getPoids() {  
        return super.getPoids() + SURPOIDS;  
    }  
}
```

## La classe Destructeur

# Exercice (solutions)

## La classe Transporteur 1/2

```
/**
 * Cette classe représente un type de robot particulier.
 * Ces robots ont la capacité de transporter un objet.
 *
 * @version oct. 2008
 * @author Stéphane Lopes
 */
class Transporteur extends Robot {
    /** Surpoids des robots transporteurs */
    public static final int SURPOIDS = 1;

    /** L'objet transporté. */
    private ObjetTransportable charge;

    /**
     * Initialise le robot.
     * @param terrain terrain sur lequel le robot va évoluer.
     * @param depart position de départ du robot.
     * @param direction orientation du robot.
     */
    public Transporteur(Terrain terrain, Position depart, Direction orientation) {
        super(terrain, depart, orientation);
    }
}
```

## La classe Transporteur (part. 1)



# Exercice (solutions)

## La classe Transporteur 2/2

```
/**
 * Retourne le poids du transporteur
 * éventuellement chargé.
 * @return le poids de l'élément.
 */
@Override
public int getPoids() {
    return super.getPoids() +
        SURPOIDS +
        (charge != null ? charge.getPoids() : 0);
}
```

## La classe Transporteur (part. 2)

# Exercice (solutions)

## L'application

```
// Création du terrain
Terrain terrain = new Terrain(4, 5, null);

// Création des éléments mobiles
final int NB_ELEMENTS = 5;
ElementMobile[] elements = new ElementMobile[NB_ELEMENTS];
elements[0] = new Robot(terrain, new Position(0, 0), Direction.EST);
elements[1] = new Transporteur(terrain, new Position(1, 1),
    Direction.EST);
elements[2] = new Destructeur(terrain, new Position(3, 0),
    Direction.SUD);
elements[3] = new ObjetTransportable(terrain,
    new Position(3, 4), 5);
elements[4] = new ObjetTransportable(terrain,
    new Position(1, 2), 2);

// Calcul du poids total
int poidsTotal = 0;
for (int i = 0; i < NB_ELEMENTS; ++i) {
    poidsTotal += elements[i].getPoids();
}
assert poidsTotal == (1+3+2+5+2): poidsTotal;
```

## L'application

# Chapitre V : Héritage en Java

## Section 17 : Classe abstraite

- Classe abstraite en Java
- La classe Number

# La classe Number et les classes ADAPTATEUR

- La classe Number est une classe abstraite de la librairie Java
- Elle définit le comportement commun aux classes pour la gestion des nombres (les conversions)
- Elle possède plusieurs sous-classes
  - les classes ADAPTATEUR : Byte, Double, Float, Integer, Long, Short
  - BigDecimal, BigInteger
- Ces classes offrent une vue « objet » des types primitifs
- Il existe d'autres classes ADAPTATEUR : Boolean, Character, Void
- Les sous-classe de Number sont des exemples du pattern de conception ADAPTATEUR

## Remarque

La plupart des fonctions arithmétiques sont des méthodes de classe de la classe Math.

# autoboxing/autounboxing

- Ce mécanisme permet d'éviter la conversion manuelle entre type primitif et classe ADAPTATEUR
- C'est simplement une facilité d'écriture (*sucre syntaxique*)

## Avant JDK 1.5

```
Integer i = Integer.valueOf(12); // préférable à new Integer(12)
int n = i.intValue();
```

## A Partir du JDK 1.5

```
Integer i = 12;
int n = i;
```

# Chapitre V : Héritage en Java

## Section 18 : Interface

- Définition
- Interface en Java

# Chapitre V : Héritage en Java

## Section 18 : Interface

- Définition
- Interface en Java

# Interface

## Du point de vue des types

- Une *interface* regroupe uniquement des signatures d'opérations et des déclarations de constantes mais aucune implémentation
- Une interface permet donc de définir un type
- Les interfaces peuvent être organisées en hiérarchies d'héritage
- Un lien d'héritage entre interface est une relation de sous-typage



# Interface

## Du point de vue des services

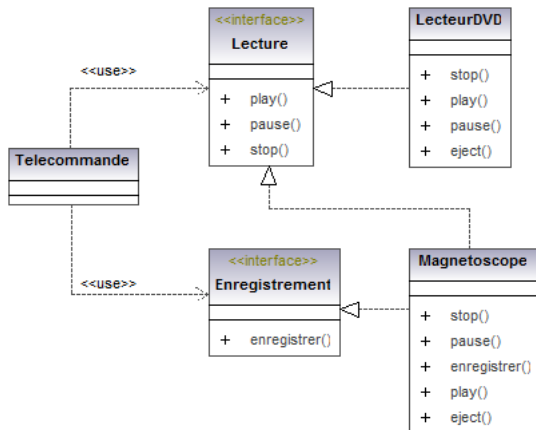
- Une *interface* fournit une vue totale ou partielle d'un ensemble de services offerts par une classe (un composant)
- Une interface est analogue à un protocole de comportement (un contrat sur un comportement)
- Une classe peut *implémenter* une ou plusieurs interfaces
- Une interface est formellement équivalente à une classe abstraite ne possédant que des méthodes abstraites

# Utilisation d'une interface

- Permet à des objets d'interagir même s'ils ne sont pas en relation
- Permet de capturer des similarités entre classes non reliées sans définir artificiellement une relation
- Permet de déclarer des méthodes qu'une ou plusieurs classes doivent implémenter
- Permet de révéler l'interface de programmation d'un objet sans révéler sa classe

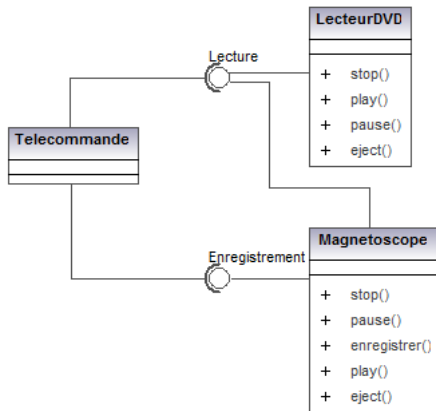
# Exemple

## Interface et implémentation d'interface



# Exemple

## Interface et implémentation d'interface



# Chapitre V : Héritage en Java

## Section 18 : Interface

- Définition
- Interface en Java

# Définition d'une interface

- La définition d'une interface comporte une déclaration et un corps

```
interface UneInterface extends UneSecondeInterface, UneAutreInterface {  
    final String uneChaine = "abcde";  
    final double unDouble = 123.456;  
    void uneMethode(int unEntier, String uneChaine);  
}
```

- Une interface peut avoir plusieurs super-interfaces
- Toutes les méthodes de l'interface sont implicitement public et abstract
- Toutes les constantes de l'interface sont implicitement public, static, et final

# Utilisation d'une interface

- Pour déclarer une classe qui implémente une ou plusieurs interfaces, on ajoute `implements ListeInterfaces` dans sa déclaration (après la clause `extends` si elle existe)
- La classe doit alors implémenter toutes les méthodes de l'interface ou être déclarée abstraite
- Une interface est utilisée comme un type (par exemple comme paramètre d'une méthode)

# Exemple

Définir l'*ordre naturel* des objets : l'interface Comparable

```
// Comparable est une interface de la librairie Java

/**
 * This interface imposes a total ordering on the objects
 * of each class that implements it. ...
 */
interface Comparable<T> {
    /**
     * Compares this object with the specified object for order. ...
     */
    public int compareTo(T o);
}

// ...
class Rectangle2D implements Comparable<Rectangle2D> {
    // ...
    public int compareTo(Rectangle2D o) {
        // Code pour la comparaison
    }
    // ...
}
```



## Chapitre VI : Module en Java

- 19 Relations entre classes
- 20 Modules
- 21 Utilisation d'une bibliothèque tierce

# Chapitre VI : Module en Java

## Section 19 : Relations entre classes

- Introduction
- Association
- Relation de dépendance

# Chapitre VI : Module en Java

## Section 19 : Relations entre classes

- Introduction
- Association
- Relation de dépendance

# Quatre types de relations entre classes

**Association** relation structurelle

**Spécialisation/généralisation** relation d'héritage ou de sous-typage (voir le chapitre précédent)

**Réalisation** relation entre une classe et une interface (voir le chapitre précédent)

**Dépendance** liaison limitée dans le temps entre objets (non structurelle)

# Chapitre VI : Module en Java

## Section 19 : Relations entre classes

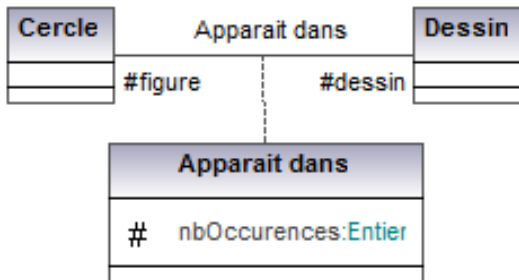
- Introduction
- Association
- Relation de dépendance

# Association

- Une *association* représente une connexion sémantique bidirectionnelle entre une (*association réflexive*) ou plusieurs classes
- L'*arité* d'une association représente le nombre de participants à l'association (association *binaire*, *ternaire*, *n-aire*)

# Exemple 1/4

Une association binaire



# Exemple 2/4

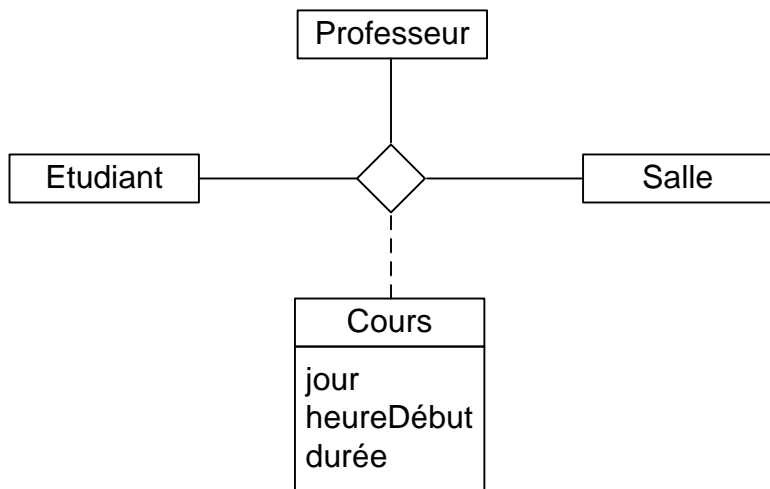
## Représentation de l'association binaire en Java

```
class ApparaîtDans {  
    private Cercle figure;  
    private Dessin dessin;  
    private int nbOccurrences;  
  
    public ApparaîtDans(Cercle figure, Dessin dessin, int nbOccurrences) {  
        this.figure = figure;  
        this.dessin = dessin;  
        this.nbOccurrences = nbOccurrences;  
    }  
    // ...  
}
```



# Exemple 3/4

Une association ternaire



# Exemple 4/4

## Représentation de l'association ternaire en Java

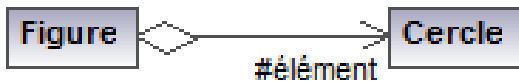
```
class Cours {  
    private Etudiant etudiant;  
    private Professeur professeur;  
    private Salle salle;  
    private int jour;  
    private int heureDebut;  
    private int duree;  
    // ...  
}
```

# Agrégation

- L'*agrégation* est une association non symétrique, qui exprime un couplage fort et une relation de subordination
- Elle représente une relation de type « ensemble/élément » (ou « tout/partie ») entre des classes
- La classe représentant l'ensemble est parfois appelée *agrégat*
- À un même moment, une instance d'élément agrégé peut être liée à plusieurs agrégats (l'élément agrégé peut être partagé)
- Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants

# Exemple 1/2

## Agrégation



# Exemple 2/2

## Représentation de l'agrégation en Java

```
class Figure {  
    /** L'ensemble de cercles composants la figure. */  
    private List<Cercle> elements;  
    // ...  
}
```

# Composition

- Une *composition* est une agrégation forte (agrégation par valeur)
- À un même moment, une instance de composant ne peut être liée qu'à un seul agrégat
- Les cycles de vies des éléments (les « composants ») et de l'agrégat sont liés

## Remarque

- Le choix de modélisation entre agrégation et composition est souvent subjectif

# Exemple 1/2

## Composition



# Exemple 2/2

## Représentation de la composition en Java

```
class Cercle {  
    /** Le centre du cercle. */  
    private Point centre;  
  
    public Cercle(final Point centre, final double rayon) {  
        this.centre = centre.clone();  
        // ...  
    }  
    // ...  
}
```



# Chapitre VI : Module en Java

## Section 19 : Relations entre classes

- Introduction
- Association
- Relation de dépendance

# Relation de dépendance

- Un élément A dépend d'un élément B si A utilise les services de B
- Un changement dans B peut donc avoir des répercussions sur A
- Par exemple, une *objet local* à une *méthode* ou un *paramètre de méthode* sont génèrent des dépendances

# Chapitre VI : Module en Java

## Section 20 : Modules

# Définition d'un module

- Pour créer un module ou y ajouter une classe ou une interface, on place une instruction `package` au début du fichier source

```
package monpackage;
```

- Tout ce qui est défini dans le fichier source fait alors parti du module
- Sans instruction de ce type, les éléments se trouvent dans le module par défaut (non nommé)
- Les noms des module respectent en général une convention (par exemple, `uvsq.in404.monpackage`)
- La librairie Java est organisée en module (`java.lang`, `java.util`, `java.io`, ...)

# Interface d'un module

- Seul les éléments publics sont accessibles à l'extérieur du module
- Pour rendre une classe ou une interface publique, on spécifie le mot-clé `public` dans sa déclaration

```
public class MaClasse { // ...
```

# Utilisation d'un module

- Différentes façons d'utiliser les éléments public d'un module
  - utiliser son nom qualifié (par exemple, `uvsq.in404.monpackage.MaClasse`)
  - importer l'élément (par exemple, `import uvsq.in404.monpackage.MaClasse`; placé en début de fichier source)
  - importer le module complet (par exemple, `import uvsq.in404.monpackage.*`; placé en début de fichier source)
  - importer les classes imbriquées (`import uvsq.in404.monpackage.MaClasse.*`;) )
  - importer les membres de classes (`import static uvsq.in404.monpackage.MaClasse.*`;) )
- Les directives `import` se placent avant toute définition de classes ou d'interfaces mais après l'instruction `package`
- Deux modules sont automatiquement importés : le module par défaut et `java.lang`

# Module et gestion des sources en Java

- Dans un fichier source
  - plusieurs éléments (classes, interfaces, ...) peuvent être définies
  - un seul élément peut être public
  - le nom de l'élément public doit être le même que le nom du fichier
- On se limite de préférence à une classe par fichier source
  - le nom du fichier `.java` est le même que le nom de l'élément qu'il contient
- Le nom du répertoire doit refléter le nom du packaging
  - la classe `uvsq.in404.monpackage.MaClasse` doit se trouver dans le fichier `MaClasse.java` du répertoire `uvsq/in404/monpackage`

# Module et compilation

- Lors de la compilation, un fichier `.class` est créé pour chaque élément
- La hiérarchie de répertoires contenant les `.class` reflète les noms des modules
- Les répertoires où sont recherchées les classes lors de l'exécution sont listés dans le *class path*
- Par défaut, le répertoire courant et la librairie Java se trouve dans le *class path*
- La façon dont le *class path* est défini dépend de la plateforme
  - en général, on définit une variable d'environnement `CLASSPATH`
- Le *class path* contient des chemins vers
  - des répertoires contenant une arborescence de `.class`
  - des fichiers `.jar`
  - des fichiers `.zip`



# Chapitre VI : Module en Java

## Section 21 : Utilisation d'une bibliothèque tierce

# Écosystème Java et bibliothèques

- L'écosystème Java fournit un nombre important de bibliothèques et d'outils de développement
- Dans un projet de développement logiciel, le choix des bibliothèques à utiliser est une étape importante
  - fonctionnalités, complexité, support de la communauté, licence, ...
- La plupart des programmes Java font appel à des *bibliothèques tierces* (*third party libraries*)

# Utilisation d'un bibliothèque tierce

- ① Récupérer la bibliothèque
  - manuellement (téléchargement)
  - automatiquement (outils de gestion des dépendances comme *maven* ou *gradle*)
- ② Inclure la bibliothèque dans le projet
  - le CLASSPATH doit être modifié pour faire référence aux archives (jar en général) de la bibliothèque
- ③ Consulter l'interface de la bibliothèque
  - toute bibliothèque Java est distribuée avec sa documentation au format *javadoc*
- ④ Importer les modules nécessaires dans les fichiers sources
  - l'utilisation d'une classe de la bibliothèque nécessite d'importer le package Java adéquat

# Exemple 1/4

Utilisation de la bibliothèque *Apache Commons Math*

- Télécharger et décompresser le fichier

[commons-math3-3.4.1-bin.tar.gz](#)

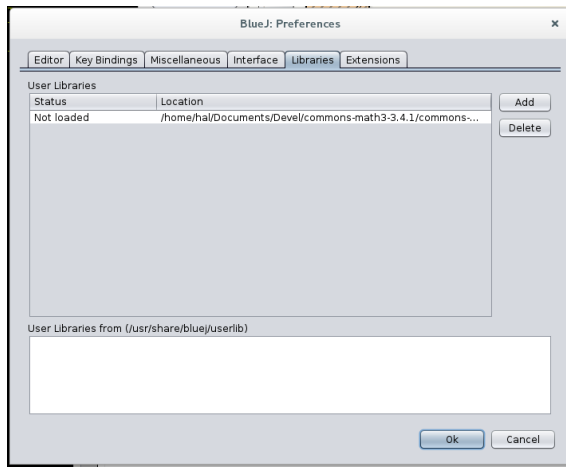
```
~/commons-math3-3.4.1 $ ls
```

commons-math3-3.4.1.jar	docs/	NOTICE.txt
commons-math3-3.4.1-javadoc.jar	LICENSE.txt	RELEASE-NOTE

## Exemple 2/4

Utilisation de la bibliothèque *Apache Commons Math*

- Ajouter la bibliothèque au projet (IDE ou outil de *build*)



# Exemple 3/4

## Utilisation de la bibliothèque *Apache Commons Math*

- Importer les classes des packages nécessaires

```
import org.apache.commons.math3.fraction.Fraction;

public class Main {
    public static void main(String[] args) {
        Fraction f = new Fraction(1, 3);
        System.out.println(f);
    }
}
```

## Exemple 4/4

Utilisation de la bibliothèque *Apache Commons Math*

- Compiler en précisant la bibliothèque dans le CLASSPATH (en ligne de commande)

```
$ javac -cp ../commons-math3-3.4.1/commons-math3-3.4.1.jar
```

- Exécuter en précisant la bibliothèque dans le CLASSPATH (en ligne de commande)

```
$ java -cp ../commons-math3-3.4.1/commons-math3-3.4.1.jar:
```

## Chapitre VII : Conclusion

### 22 Début de la conclusion



# Chapitre VII : Conclusion

## Section 22 : Début de la conclusion

# Bilan

- 1 blabla

# Pour aller plus loin. . .

- 1 blabla