Stage de Programmation Objet

LP DAWIN

Informatique

Partie 3

Héritage

Partie 3 : Héritage

Pourquoi l'héritage ?

- Dans la Partie 2, nous avons vu que Java permet plusieurs implémentations d'un même type (via les interfaces).
- C'est un principe de POO en général.
- Il faut donc (aussi) des moyens de réutiliser du code.
- L'un des moyens en POO est l'héritage.

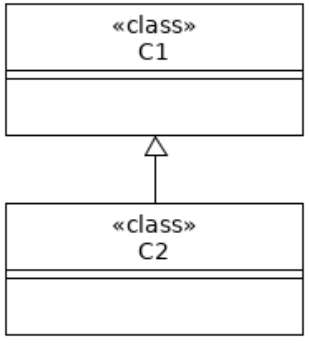
Héritage de classe

Si une classe `C2` *hérite de* `C1`*, alors tout ce qui est défini dans* `C1` *l'est aussi dans* `C2`*.*
(défini, mais pas forcément accessible, cf `private`*)*

- syntaxe :

```
class C1 { ... }
class C2 extends C1 { ... }
```

- on dit alors que :
  - `C2` est une classe fille (ou sous-classe) de `C1` (subclass)
  - `C1` est la classe mère de `C2` (superclass)
- en UML :



Héritage d'interface

→ C'est exactement la même chose pour les interfaces.

Si une interface `I2` *hérite de* `I1`*, alors tout ce qui est défini dans* `I1` *l'est aussi dans* `I2`*.*

- syntaxe :

```
interface I1 { ... }
interface I2 extends I1 { ... }
```

3 cas d'utilisation

Nous allons voir 3 cas d'utilisation classique de l'héritage :

- l'extension
- l'adaptation (spécialisation)
- la factorisation (+ classes abstraites)

Partie 3 : Héritage

Extension

On peut utiliser l'héritage pour ajouter :

- des attributs/méthodes aux classes
- des méthodes aux interfaces

```
class Salle {
  ...
  void afficher() {...}
}
```

```
class SalleLibreService extends Salle {
  Date debutOuverture;
  Date finOuverture;
  boolean ouvert(Date heure) { ... }
  ...
}
```

```
Salle s = new Salle();
s.afficher(); // ok
if (s.ouvert(h)) {...}//KO
```

```
SalleLibreService s
  = new SalleLibreService();
s.afficher(); // ok
if (s.ouvert(h)) {...} // ok
```

→ donc ajouter des données ou des fonctionnalités

Extension

On peut naturellement combiner l'héritage de classe et d'interface.

Les salles :

```
interface ISalle {
  void afficher();
}
```

```
class Salle implements ISalle {
  ...
  void afficher() {...}
}
```

Les salles en libre service :

```
interface ISalleLibreService
    extends ISalle {
  boolean ouvert(Date heure);
}
```

```
class SalleLibreService
      extends Salle
      implements ISalleLibreService {
  ...
  boolean ouvert(Date heure) { ... }
  ...
}
```

→ pas besoin de redéfinir `afficher()`

Partie 3 : Héritage

Adaptation

L'héritage de classe peut être utilisé pour obtenir de nouvelles versions de classes existantes, obtenues en redéfinissant certaines méthodes. Cela s'appelle la redéfinition de méthode (*method overriding*).

La signature doit être exactement la même. L'annotation `Override` rend explicite la redéfinition.

```
class C1 {
    ...
    void m(int i) { ... }
}
```

```
class C2 extends C1 {
    ...
    @Override
    void m(int i) { ... }
}
```

Adaptation : exemple

```
class C1 {
    void hi()  { System.out.println("Hi from C1." ); }
    void bye() { System.out.println("Bye from C1."); }
}
```

```
class C2 extends C1 {
    @Override
    void hi() { System.out.println("Hi from C2."); }
}
```

```
C1 c1 = new C1();
C2 c2 = new C2();
c1.hi(); // prints "Hi from C1."
c2.hi(); // prints "Hi from C2."
c1.bye(); // prints "Bye from C1."
c2.bye(); // prints "Bye from C1."
```

super

Il est possible d'appeler une méthode de la classe mère via le mot-clé `super`.

```
class Salle {
  ...
  void afficher() {...}
}
```

```
class SalleLibreService extends Salle {
  ...
  @Override
  void afficher() {
    super.afficher();
    System.out.println(
      "Heures d'ouverture : " + ...);
  }
}
```

C'est très classique dans les constructeurs :

```
class Salle {
  ...
  Salle(int etage) {
    ...
  }
  ...
}
```

```
class SalleLibreService extends Salle {
  ...
  SalleLibreService(int etage, Date o, Date f) {
    super(etage);
    debutOuverture = o;
    finOuverture = f;
  }
  ...
}
```

→ l'appel à `super()` est même fait implicitement dans les constructeurs, s'il n'y a pas eu surcharge.

Partie 3 : Héritage

Factorisation : cas simple

Imaginons 2 classes avec une même méthode `m()` :

```
class C1 {
  ...
  void m() { ... }
  ...
}
```

```
class C2 {
  ...
  void m() { ... }
  ...
}
```

L'héritage permet de la placer dans une classe mère `C` :

```
class C {
  ...
  void m() { ... }
  ...
}
```

```
class C1 extends C {
  ...
}
```

```
class C2 extends C {
  ...
}
```

Factorisation : cas moins simple

Parfois, ce n'est pas si simple.

```
interface ISalle {
  void afficherPlan();
  boolean salleMachine();
}
```

```
class SalleTD implements ISalle {
  void afficherPlan() { ... }
  boolean salleMachine() {
    return false;
  }
}
```

```
class SalleTP implements ISalle {
  void afficherPlan() {...} // idem SalleTD
  boolean salleMachine() {
    return true;
  }
}
```

ce qui donnerait (en gardant `ISalle`) :

```
class Salle {
  void afficherPlan() { ... }
}
```

```
class SalleTD
    extends Salle
    implements ISalle {
  boolean salleMachine() {
    return false;
  }
}
```

```
class SalleTP
    extends Salle
    implements ISalle {
  boolean salleMachine() {
    return true;
  }
}
```

→ problème : on ne peut pas avoir `Salle implements ISalle`.

Factorisation : classes abstraites

→ solution : une classe abstraite !

- certaines méthodes peuvent être simplement déclarées, sans avoir de code associé.
  - on les appelle méthodes abstraites
  - on les déclare via le mot-clé `abstract`.
- une classe abstraite est simplement une classe, dont au moins une des méthodes est abstraite.
  - on les déclare aussi via le mot-clé `abstract`.

(Une classe non-abstraite est dite concrète.)

Factorisation : cas moins simple (solution)

```
interface ISalle {
  void afficherPlan();
  boolean salleMachine();
}
```

```
class SalleTD implements ISalle {
  void afficherPlan() { ... }
  boolean salleMachine() {
    return false;
  }
}
```

```
class SalleTP implements ISalle {
  void afficherPlan() {...} // idem SalleTD
  boolean salleMachine() {
    return true;
  }
}
```

Désormais :

```
abstract class Salle implements ISalle {
  void afficherPlan() { ... }
  abstract boolean salleMachine();
}
```

```
class SalleTD extends Salle {
  boolean salleMachine() {
    return false;
  }
}
```

```
class SalleTP extends Salle {
  boolean salleMachine() {
    return true;
  }
}
```

Factorisation : classes abstraites

```
abstract class Salle implements ISalle {
  void afficherPlan() { ... }
  abstract boolean salleMachine();
}
```

Une classe abstraite :

- ne peut pas être instanciée,

```
Salle s = new Salle(); // impossible
```

- peut contenir des attributs et des constructeurs (comme toute classe),
- définit un type de données (comme toute classe).

```
Salle s = new SalleTD(); // ok
```

Remarque : une interface est une classe abstraite pure sans attribut, c'est-à-dire contenant uniquement des méthodes abstraites.

Partie 3 : Héritage

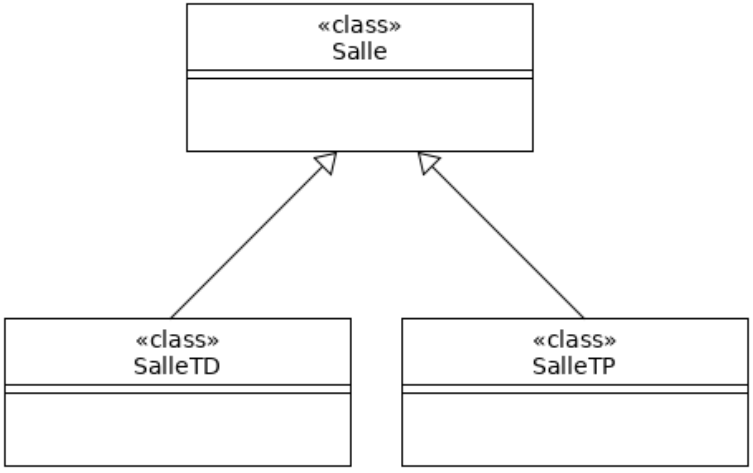
Règles sur l'héritage et l'implémentation (1/3)

- une interface *peut* être implémentée par plusieurs classes.

```
interface ISalle { ... }
class SalleTD implements ISalle { ... }
class SalleTP implements ISalle { ... }
```

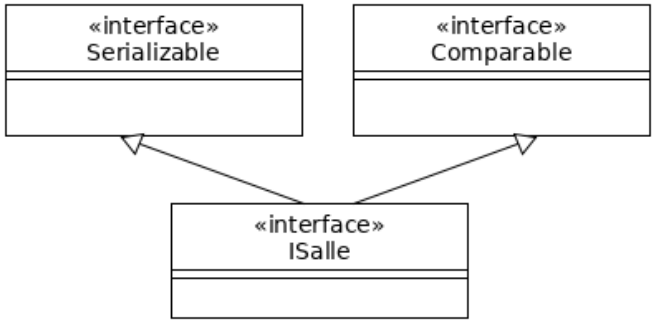
- une classe *peut* être héritée par plusieurs classes.

```
class Salle { ... }
class SalleTD extends Salle { ... }
class SalleTP extends Salle { ... }
```



Règles sur l'héritage et l'implémentation (2/3)

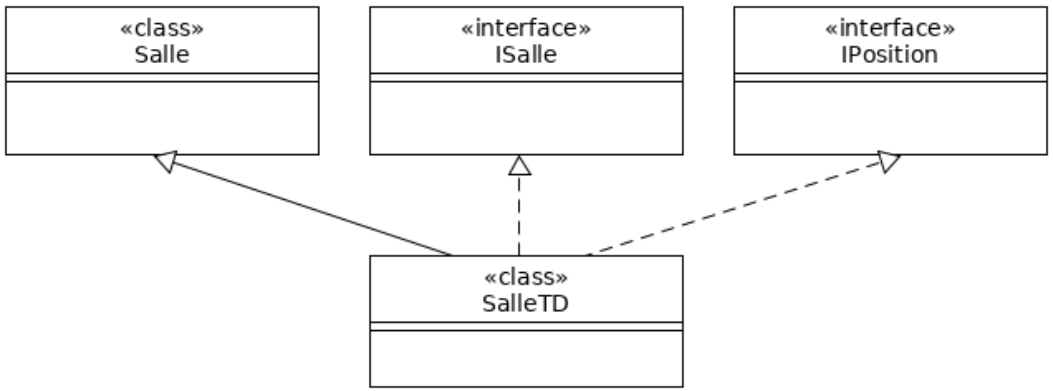
- une interface *peut* hériter de plusieurs interfaces.




```
interface ISalle extends Serializable, Comparable { ... }
```

- une classe *peut* implémenter plusieurs interfaces, et même hériter en même temps :

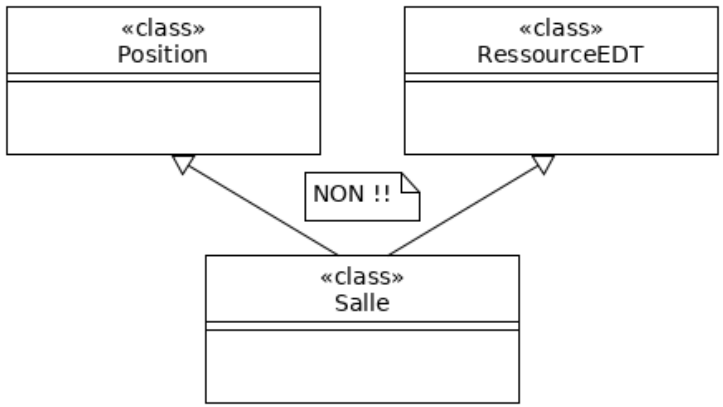
```
class Salle { ... }
class SalleTD extends Salle implements ISalle, IPosition { ... }
```



Règles sur l'héritage et l'implémentation (3/3)

- une classe *ne peut pas* hériter de plusieurs classes.

```
class Position { ... }
class RessourceEDT { ... }
class Salle extends Position, RessourceEDT { ... } // interdit !
```



- on s'en sort parfois en utilisant la délégation :

```
class Salle extends Position {
  RessourceEDT ressource;
  ...
}
```

autre contournement possible depuis Java 8 : méthodes par défaut des interfaces (c'est Mal : dénature les interfaces)

L'API Java

L'API (Application Programming Interface) de Java, autrement dit la bibliothèque standard, est très riche. C'est l'un des avantages de Java.
Tout développeur doit s'y référer, cf documentation.

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

→ Le package java.lang est importé par défaut. Pour les autres : `import`.

L'API Java : Object

On remarquera en particulier qu'en Java tout objet hérite (directement ou indirectement) de la classe `Object`.

Partie 3 : Héritage

Mini-projet : v4 $\longrightarrow$ v5

Niveau 1

- on souhaite désormais pouvoir faire jouer un humain contre une IA qui joue aléatoirement. Dans "fill" l'IA sera le joueur blanc, dans "crush" le joueur noir.
- définir une classe abstraite `Player`, avec des sous-classes `HumanPlayer` et IAPlayer
- `Action.getFromPlayer()` devient `Player.play()`
- propager partout...

Niveau 2

- on ajoutera une classe abstraite `Game`.

Partie 3 : Héritage

Polymorphisme

polymorphisme $\neq$ métamorphose

quoique...

$\rightarrow$ Le polymorphisme, c'est :

- la possibilité d'associer plusieurs types à un même objet...
- et d'en tirer profit, en appliquant un même traitement à des objets de types différents.

Un objet, plusieurs types

Comment un objet peut-il avoir plusieurs types ?

Déjà vu :

```
interface ISalle { ... }
class Salle implements ISalle { ... }
ISalle s = new Salle(...);
```

En vrai, en Java,

- une variable possède un type, et

```
ISalle s;
```

- une instance possède un type (potentiellement différent).

```
s = new Salle(...);
```

- ces types ne changeront jamais

Un objet, plusieurs types

Autrement dit :

- chaque instance a son type réel
- chaque instance a un type apparent : celui de la variable (ou du paramètre) à laquelle elle est assignée

→ chaque instance peut être "*vue comme*" d'un autre type :

une `Salle` :

```
Salle s = new Salle();
```

cette salle "*vue comme*" une `ISalle` :

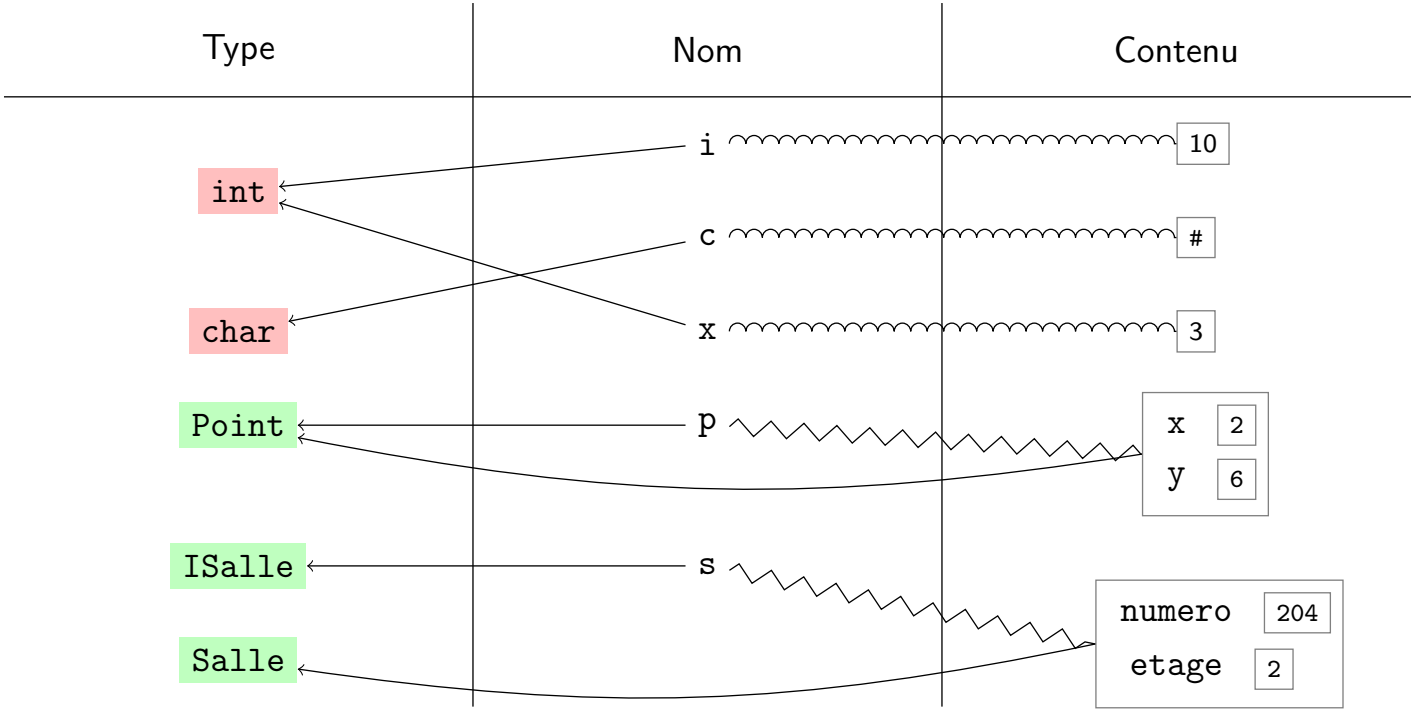
```
ISalle i = s;
```

Tony Stark :

Tony Stark "*vu comme*" IronMan :

Un objet, plusieurs types

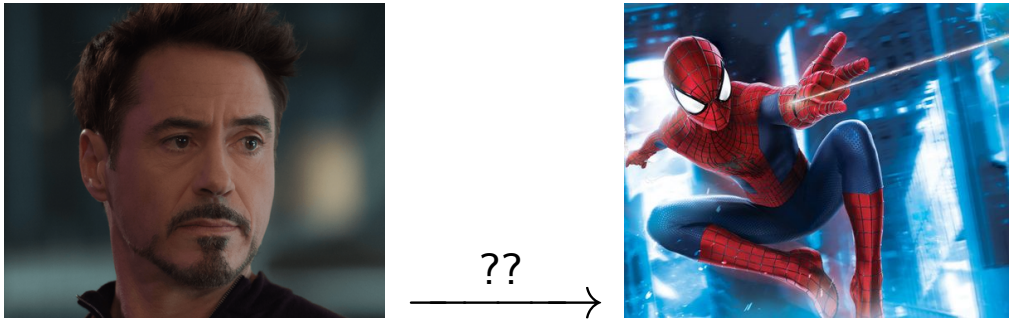
Donc le typage ressemble plutôt à ça :




```
Point p = new Point();
...
interface ISalle { ... }
class Salle implements ISalle { ... }
ISalle s = new Salle(...);
```

Un objet, plusieurs types

- Alors peut-on assigner n'importe quel type apparent à un objet ?




```
ISalle s = new Etudiant(); // ??
```

→ NON !

Un objet, plusieurs types

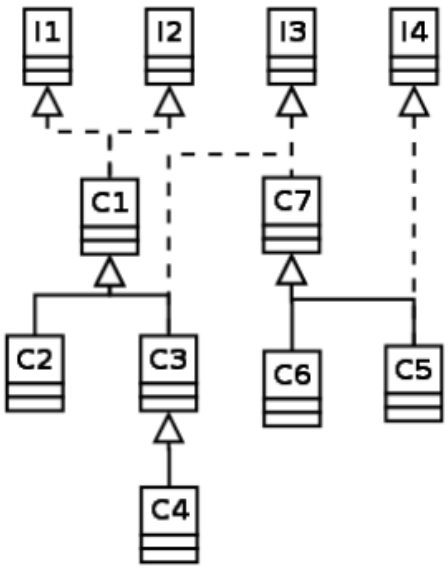
Le type réel d'un objet doit être :

- son type apparent, ou
- un de ses sous-types.

Un type `T2` est un sous-type de `T1` si :

- `T2` hérite de `T1` (si `T1` est une classe)
  - soit directement soit en suivant une suite de liens d'héritage, ou
- `T2` implémente `T1` (si `T1` est une interface)
  - ou alors `T2` hérite (directement ou pas) d'une classe implémentant `T1`

Un objet, plusieurs types



→ le type réel doit être plus spécifique (au sens large) que le type apparent : c'est la relation "est un" :

- une salle TD est une salle, mais
- une salle n'est pas (forcément) une salle TD

```
class Salle { ... }
class SalleTD extends Salle { ... }
Salle s = new SalleTD(); // OK
SalleTD std = new Salle(); // Non !
```

Partie 3 : Héritage

Le polymorphisme en action

```java
interface ISalle {
  int capacite();
  int nbOrdis();
}
```

```java
abstract class Salle
    implements ISalle {
  private final int capacite;

  Salle(int capa) {
    capacite = capa;
  }

  @Override
  int capacite() {
    return capacite;
  }

  @Override
  abstract int nbOrdis();
}
```

```java
class SalleTD extends Salle {
  @Override
  int nbOrdis() { return 1; }
}
```

```java
class SalleTP extends Salle {
  @Override
  int nbOrdis() {
    return 1 + capacite()/2;
  }
}
```

Exemple de polymorphisme :

```java
ISalle[] salles = new ISalle[3];
salles[0] = new SalleTP(26);
salles[1] = new SalleTD(60);
salles[2] = new SalleTP(26);
int nbOrdisSalles = 0;
for (ISalle s : salles) {
  nbOrdisSalles += s.nbOrdis();
}
```

$\rightarrow$ polymorphisme qu'on pourrait appeler "horizontal"

Le polymorphisme en action

Autre implémentation, avec nbOrdis par défaut :

```
interface ISalle {
  int capacite();
  int nbOrdis();
}
```

```
class Salle
    implements ISalle {
  private final int capacite;

  Salle(int capa) {
    capacite = capa;
  }

  @Override
  int capacite() {
    return capacite;
  }

  @Override
  int nbOrdis() { return 1; }
}
```

```
class SalleTD extends Salle {
}
```

```
class SalleTP extends Salle {
  @Override
  int nbOrdis() {
    return 1 + capacite()/2;
  }
}
```

Même exemple :

```
ISalle[] salles = new ISalle[3];
salles[0] = new SalleTP(26);
salles[1] = new SalleTD(60);
salles[2] = new SalleTP(26);
int nbOrdisSalles = 0;
for (ISalle s : salles) {
  nbOrdisSalles += s.nbOrdis();
}
```

$\longrightarrow$ polymorphisme qu'on pourrait appeler "vertical"

Le polymorphisme en action

Autre implémentation, avec nbOrdis par défaut :

```
interface ISalle {
  int capacite();
  int nbOrdis();
}
```

```
class Salle
    implements ISalle {
  private final int capacite;

  Salle(int capa) {
    capacite = capa;
  }

  @Override
  int capacite() {
    return capacite;
  }

  @Override
  int nbOrdis() { return 1; }
}
```

```
class SalleTD extends Salle {
}
```

```
class SalleTP extends Salle {
  @Override
  int nbOrdis() {
    return 1 + capacite()/2;
  }
}
```

Remarque : la sélection de la méthode à appliquer est faite à l'exécution :

```
SalleTP s1 = new SalleTP(26);
SalleTD s2 = new SalleTD(60);
ISalle s;
if (r % 2 = 0) { // r entier aleatoire
  s = s1;
} else {
  s = s2;
}
System.out.println(s.nbOrdis());
```

Partie 3 : Héritage

instanceof

En Java, `s instanceof Salle` permet de tester si le type de la variable `s` est `Salle`, ou un de ses sous-types.

Par exemple (avec la version où `Salle` n'est pas abstraite) :

```
SalleTD s = new SalleTD(26);
if (s instanceof Object ) { System.out.print("Object "); }
if (s instanceof ISalle ) { System.out.print("ISalle "); }
if (s instanceof Salle  ) { System.out.print("Salle "); }
if (s instanceof SalleTD) { System.out.print("SalleTD "); }
if (s instanceof SalleTP) { System.out.print("SalleTP "); }
```

affiche : `Object ISalle Salle SalleTD`

```
Salle s = new Salle(26);
if (s instanceof Object) { System.out.print("Object "); }
... // idem
```

affiche : `Object ISalle Salle`

Transtypage

Le transtypage (*typecast*) permet de changer le type apparent d'un objet à la volée :

- soit "vers le haut" (*upcasting*). C'est souvent inutile, car fait implicitement par Java :

```
Salle s = (Salle)(new SalleTD(26));
// ou plus simple : Salle s = new SalleTD(26);
```

- soit "vers le bas" (*downcasting*), ce qui est utile quand on veut retrouver un traitement spécifique :

```
class SalleTP extends SalleTD {
  void reboot(); // uniquement dans SalleTP
}
```

```
ISalle[] salles = new ISalle[3];
salles[0] = new SalleTP(26);
salles[1] = new SalleTD(60);
salles[2] = new SalleTP(26);
int nbOrdisSalles = 0;
for (ISalle s : salles) {
  nbOrdisSalles += s.nbOrdis();
  if (s instanceof SalleTP) { // ClassCastException sinon
    ((SalleTP)s).reboot();
  }
}
```

Partie 3 : Héritage

Mini-projet : v5 → v6

Niveau 1

- ajouter le jeu "crush3" : comme crush, sauf que chaque joueur ne dispose que de 3 pions (égalité si tous les pions sont posés).
  - créer une sous-classe `LimitedGame` de `Game`, telle que le nb de pièces initiales par joueur puisse varier d'un jeu à l'autre
  - créer une sous-classe `LimitedGameState` de `GameState`, qui gère le nombre de pièces des joueurs

Niveau 2

- supprimer `ActionType`, et remplacer `Action` par une interface implémentée par 4 classes (`Play`, `Load`, `Save`, et `Quit`)

```
public interface IAction {
    public void perform(IGame game, GameState state);
}
```

Remarque : dans `Load`, que fait : `state = FileUtils.load(game);` ?