

Java et la Conception Objet

Introduction

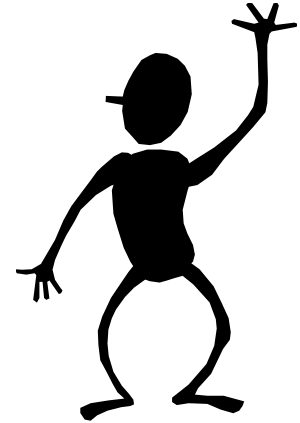
Bienvenue !

Présentation :

- de votre formateur
- de Valtech Training
- des cours associés

Présentation des stagiaires

- Expérience dans des domaines proches
- Objectifs



Votre environnement



Principaux objectifs

Réfléchir et concevoir en termes d'objets

- comprendre les concepts de base
- appliquer des principes reconnus

Connaître la syntaxe Java

Utiliser les bibliothèques Java de base

Appliquer un processus itératif pour l'analyse, la conception et la programmation orientées objet

Les concepts nouveaux seront introduits, puis repris et développés

Le tout mis en pratique au fur et à mesure au travers d'une étude de cas !

Orientation objet

Analyse orientée objet

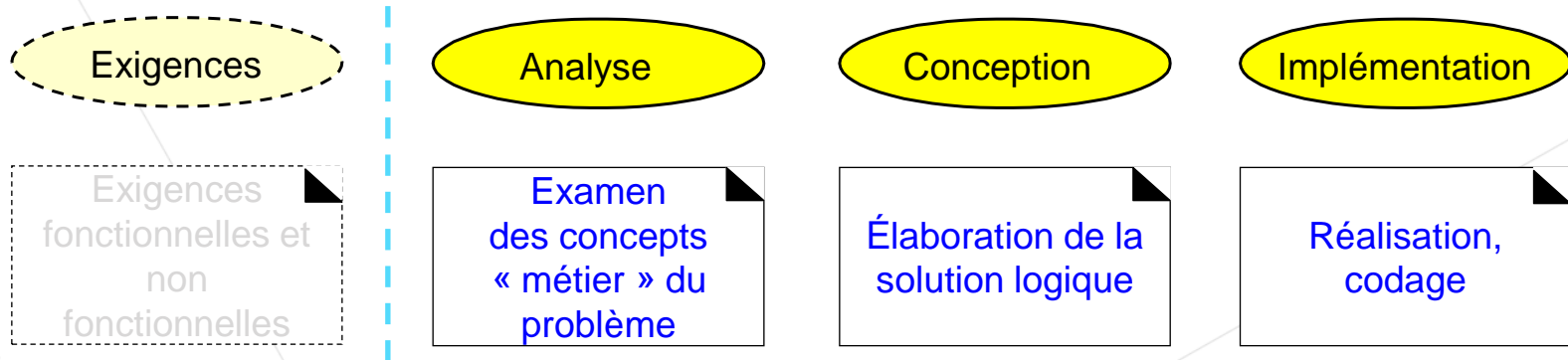
- Quels sont les objets du monde réel ?

Conception orientée objet

- Les objets logiciels ? Leurs responsabilités ?

Programmation orientée objet

- Codage dans un langage de programmation orienté objet

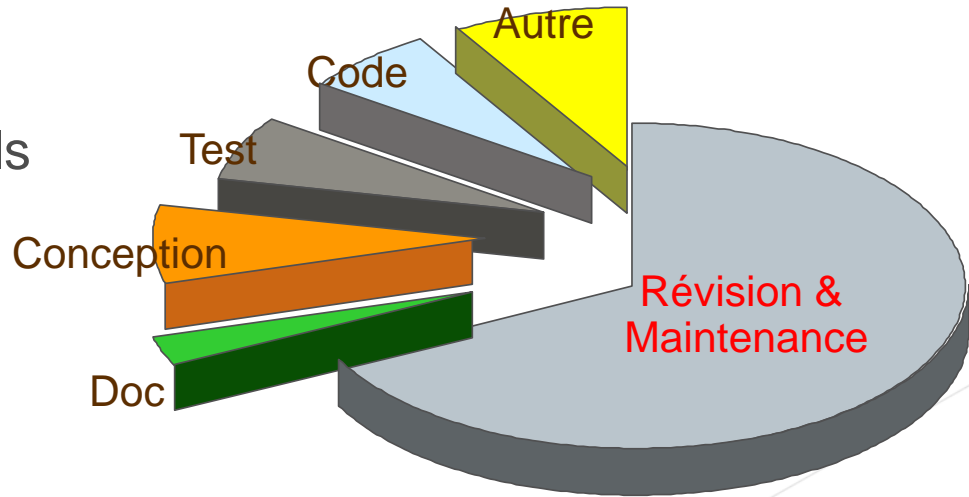


Pourquoi la technologie objet ?

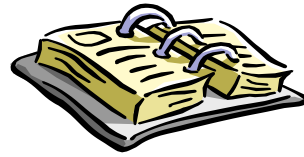
Parce que les systèmes objet sont :

- *Faciles à maintenir*
- *Extensibles*
- *Flexibles*

Coût des projets logiciels



Contenu de la formation



1. A la découverte des « objets » d'un système
2. Concepts orientés objet fondamentaux
3. Architecture de Java, outils et bibliothèques standards
4. Définition de classes en Java
5. Autres éléments de syntaxe Java
6. Structure de contrôle Java
7. Collaboration entre objets – relations, agrégation (« containment »)
8. Tableaux d'objets et de primitives
9. Les packages
10. Notion de processus de développement
11. Modèle du domaine (ou d'analyse)
12. Vers le modèle de conception
13. Modèle dynamique – diagrammes d'interactions
14. Diagramme de classes de conceptions
15. Mapping des artefacts de conception avec le code
16. Les concepts fondamentaux
17. La classe Object
18. Les interfaces
19. Les collections Java 2
20. Méthodes et variables statiques
21. Gestion des exceptions

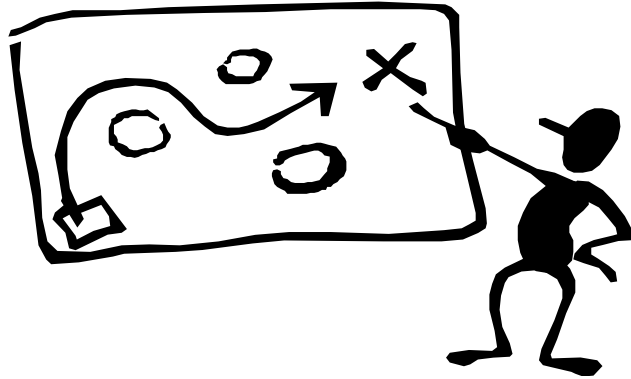
Java et la Conception Objet

**A la découverte des « objets »
d'un système**

Objectifs

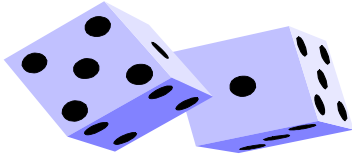
Identifier les objets logiciels d'un système simple (modélisation statique)

Examiner la façon dont ces objets collaborent pour mener à bien un scénario (modélisation dynamique)



Le Monde Réel et le Pays des Objets

Monde réel



abstraction

Monde logiciel

De
face
getFace() lancer()

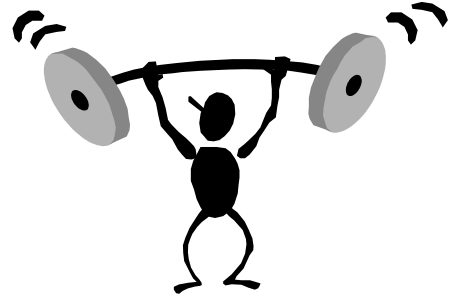
implémentation

```
public class De
{
    private int face;

    public int getFace( ) {...}
    public void lancer( ){...}
    //...
}
```

Un objet associe des données (attributs) et du code (comportement) au sein d'une même unité

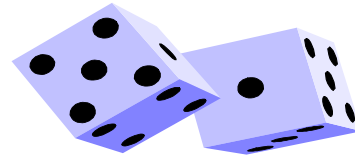
Notre projet



Nous allons mettre au point la représentation orientée objet d'un jeu de dés simple

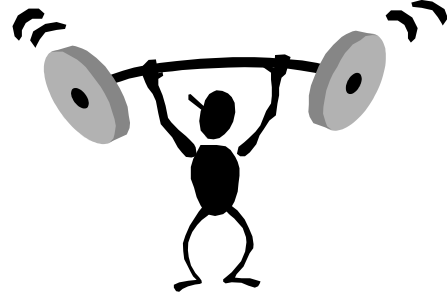
Règles :

- Le jeu se joue de 2 à n joueurs
- Chaque joueur lance les deux dés à 6 faces
- Il y a dix tours
- Le joueur tirant le plus souvent sept ou plus a gagné



Exercice : identifiez les objets du monde réel liés au jeu de dés

Attributs



Les attributs décrivent les informations dont doit se souvenir un objet

Exemples :

- Le De a une face

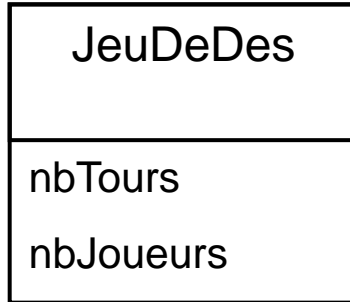
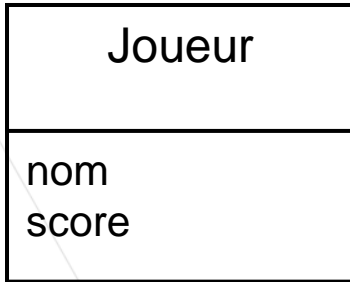
Joueur
nom
score

Exercice : identifiez les attributs de chaque objet du jeu de dés

Modèle objet

Modèle des objets et attributs importants

« Dictionnaire visuel » des concepts et du vocabulaire importants



Dynamique : responsabilités et collaborations

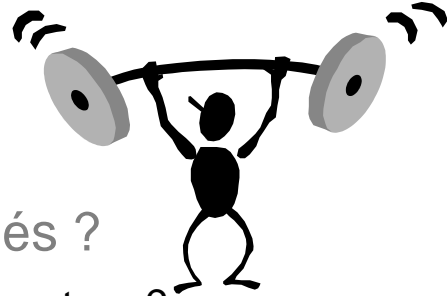
Après la modélisation statique, pensez aux objets logiciels en termes de responsabilités

- Responsabilité : savoir, ou savoir faire
- La responsabilité d'un dé est de connaître sa face

Les objets peuvent collaborer avec d'autres pour assumer leurs responsabilités

- Un dé peut collaborer avec le joueur pour se lancer ...
- Semblable à la collaboration entre personnes en fonction de leurs spécialités et de leurs connaissances

Exercice dirigé

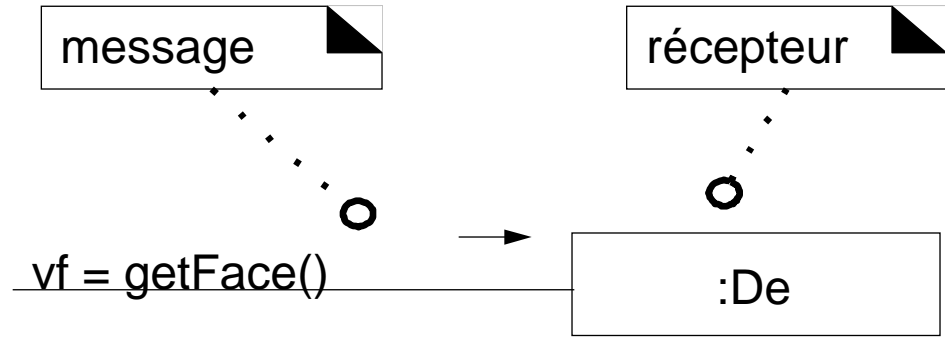


Quelles sont les responsabilités des objets du jeu de dés ?

- Quels sont les objets susceptibles de collaborer les uns avec les autres ?
- Remarquez qu'il s'agit d'une étape de conception créative avec toute une gamme de réponses
- Une part importante de ce cours consiste à explorer les questions suivantes :
« qu'est-ce qu'une bonne conception et pourquoi ? »

Collaborations et messages

La collaboration s'effectue par l'envoi de messages



Message

- Signal envoyé à un objet (récepteur) pour l'invocation d'une méthode
- Peut retourner une valeur
- Semblable à un appel de fonction, mais dirigé vers un objet

Diagramme de communication

Affiche les détails précis des collaborations en termes de messages

- un message invoque une action sur un objet
- Les objets collaborent pour résoudre des problèmes

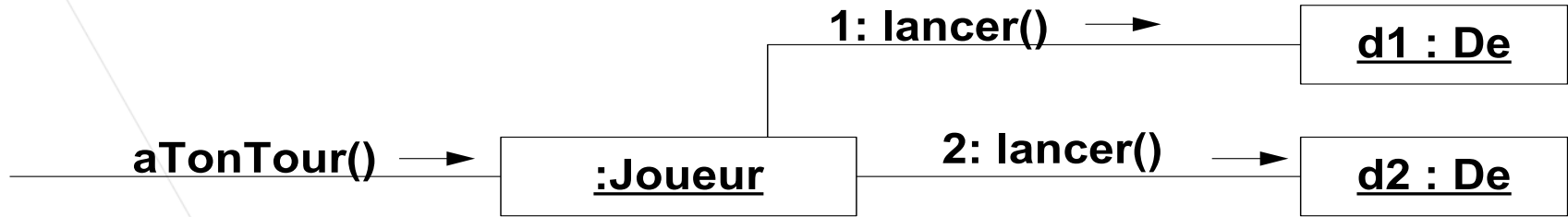
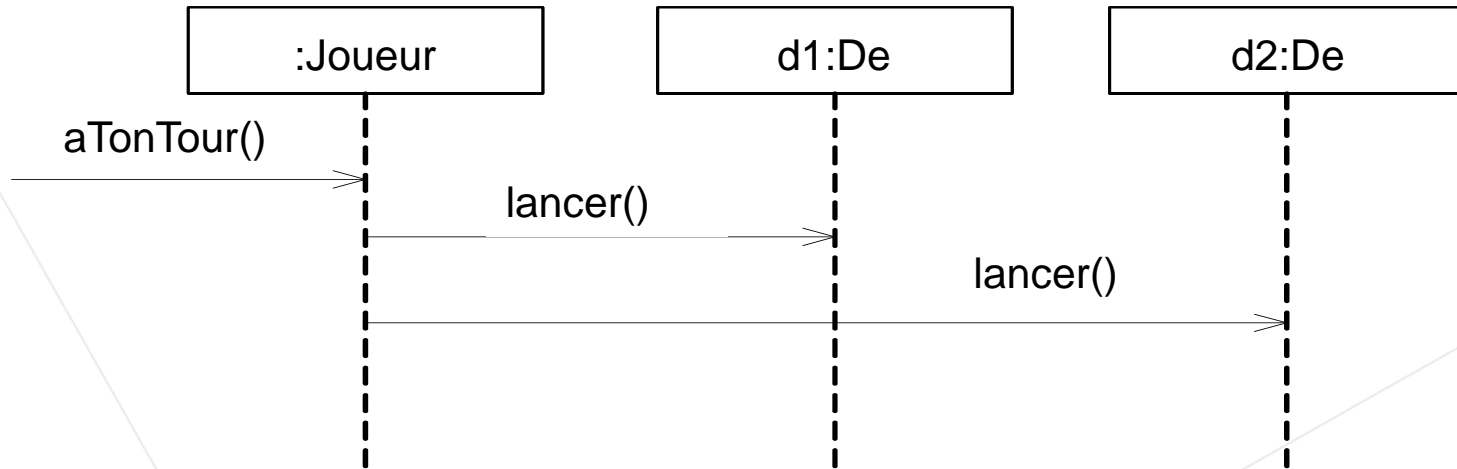


Diagramme de séquence

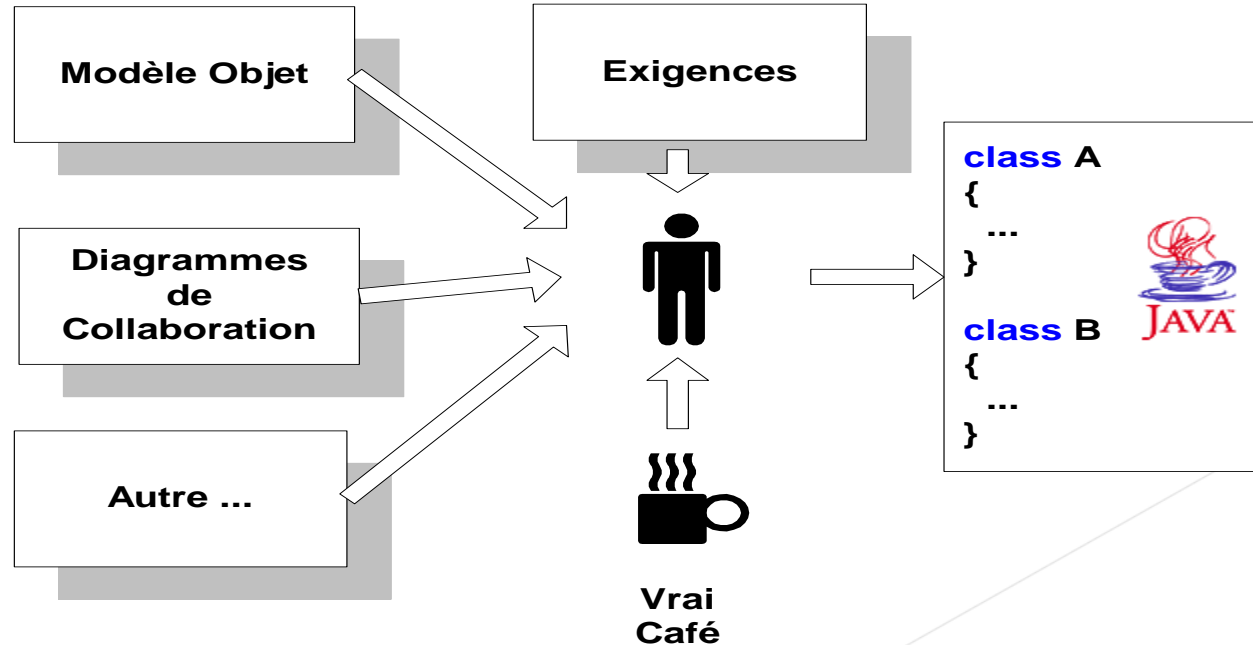
Le diagramme de séquence est exactement équivalent au diagramme de communication

- Le temps s'écoule de haut en bas
- Met l'accent sur l'ordre chronologique des messages

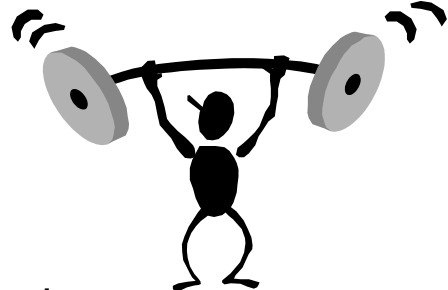


Et enfin... création d'un système Java

À l'aide de documents, de stimulants et de principes de conception objet, le code Java est développé



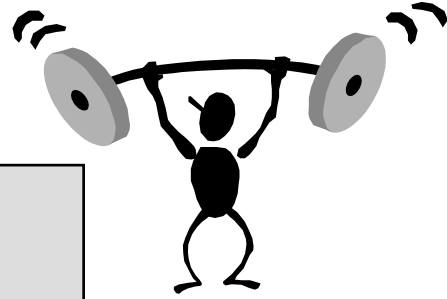
Exercice



Assez de théorie: l'heure est à la pratique !

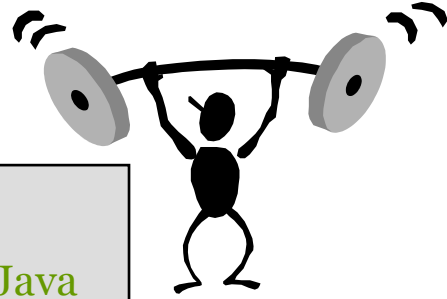
- Sous la direction du formateur, utilisez un outil de développement Java pour définir et tester un objet Dé (voir les pages suivantes) chargé de rouler et de connaître sa valeur faciale
 - Il y a un grand nombre de concepts implicites « masqués » dans cet exercice. Ceux-ci sont expliqués dans les modules suivants

Classe De — Code source



```
public class De {  
    private int face;  
  
    public int getFace() {  
        return face;  
    }  
  
    public void lancer() {  
        face = (int) (Math.random() * 6) + 1;  
    }  
}
```

Classe Main — Code source



```
public class Main {  
    // "main" est le point de départ d'exécution dans un programme Java  
    public static void main( String[] args ) {  
        // création d'un objet De  
        De d1 = new De();  
        // boucle 10 fois  
        for ( int i = 0; i < 10; i++ )    {  
            // envoi du message "lancer"  
            d1.lancer();  
            int valeur = d1.getValeurFace();  
            // impression sur la console  
            System.out.println( valeur );    }  
    }  
}
```

Test du code

Vous venez de voir un exemple de bonne pratique ; soyons maintenant plus explicites

- Remarquez que, dans l'exemple précédent, il a été créé une classe pour tester l'objet De.
- Il est courant et conseillé de créer ce style de test pour toute classe que vous définissez.
- On appelle cela des « tests unitaires » — il existe des *frameworks* (par exemple, *JUnit*) permettant d'automatiser l'exécution des tests unitaires.
- Avoir le formateur réalisez un exemple de tests unitaires.

Résumé

Les objets logiciels associent les attributs et les comportements

Les objets ont des responsabilités et collaborent par l'intermédiaire de messages pour effectuer leur travail

Différents documents, tels que modèle objet (statique) et diagrammes dynamiques (séquence, communication), sont utilisés pendant le développement



Java et la Conception Objet

Concepts orientés objet fondamentaux

Qu'est-ce qu'un objet ?

Dans la technologie objet, un objet est un composant logiciel

Un objet regroupe les données (attributs) et le code (comportement) en une même unité

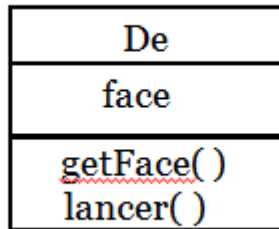
Analogie : un objet s'apparente à un enregistrement informatique (tel qu'une *struct* C ou un enregistrement *COBOL*) ayant ses propres fonctions

De
face
getFace() lancer()

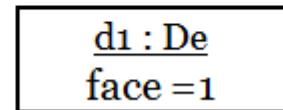
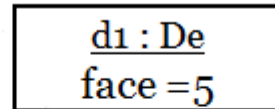
Classe et instance dans le logiciel

Une classe :

- Définit – la classe définit les caractéristiques et le comportement d'une famille d'objets (ses instances)
- Crée – c'est une usine à créer des objets logiciels d'un certain type



créer



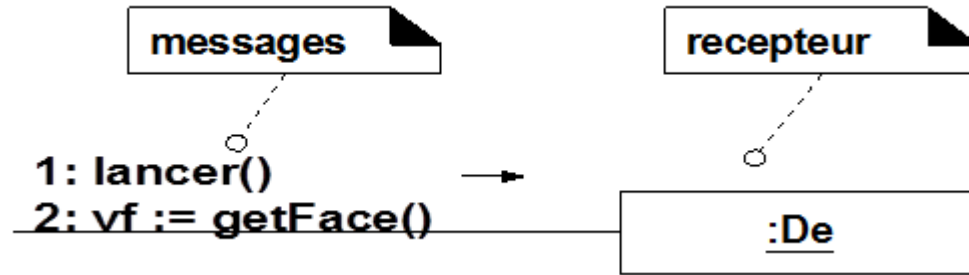
Une instance :

- Est instanciée (créée) par une classe
- Occupe de l'espace dans la mémoire d'un ordinateur
- Conserve les valeurs des attributs
- A un comportement
- Connaît sa classe

Messages et méthodes

Un message

- est un signal vers un objet (récepteur) pour invoquer une méthode
- peut retourner une valeur
- est semblable à un appel de fonction, mais dirigé vers un objet



Une méthode

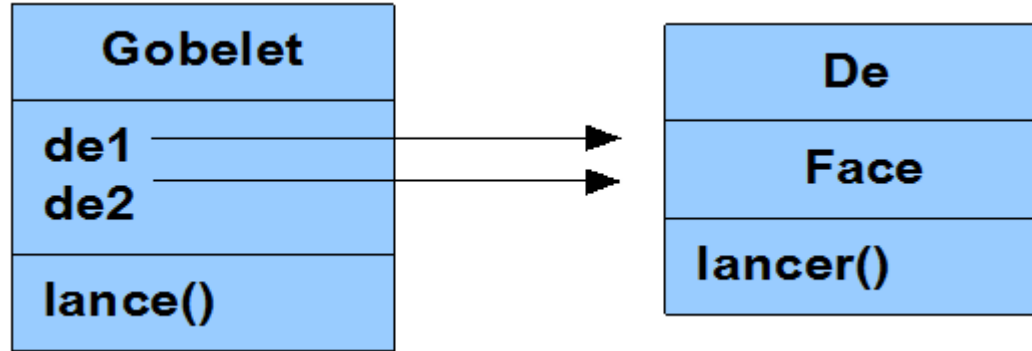
- est le comportement d'un objet invoqué par un message
- est semblable à une fonction, mais fait partie d'un objet

Contenance

Des objets peuvent en contenir d'autres

La complexité est répartie en couches

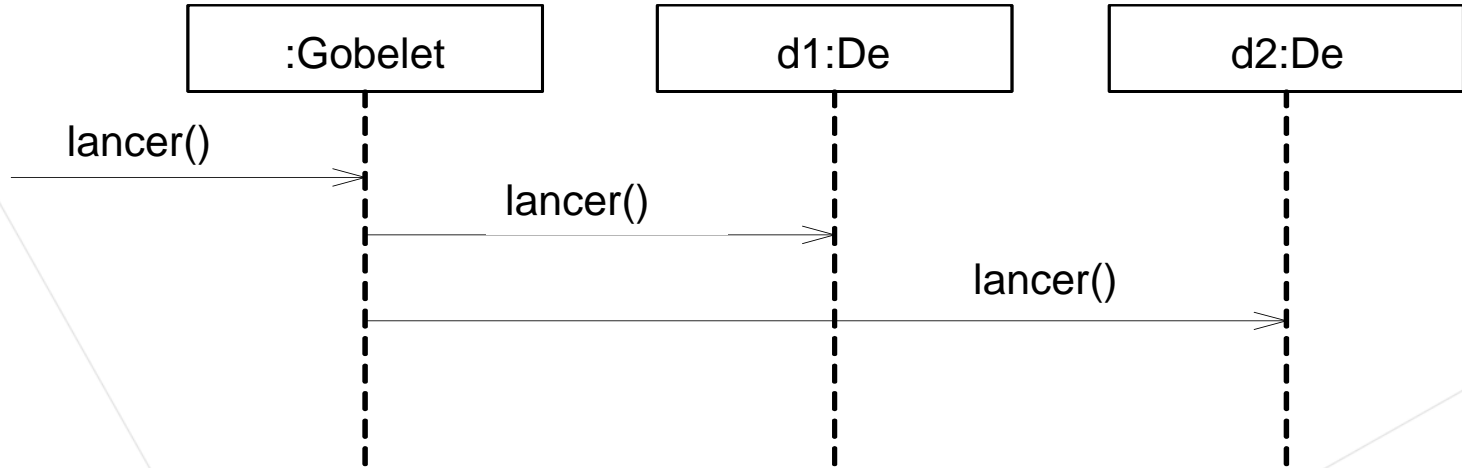
Les objets sont liés les uns aux autres



Interaction/collaboration entre objets

Collaboration pour l'accomplissement de tâches

L'un des objets est l'émetteur, l'autre est le récepteur du message



Principe fondamental 1/3: Encapsulation

L'encapsulation fait référence au masquage des informations ou des détails

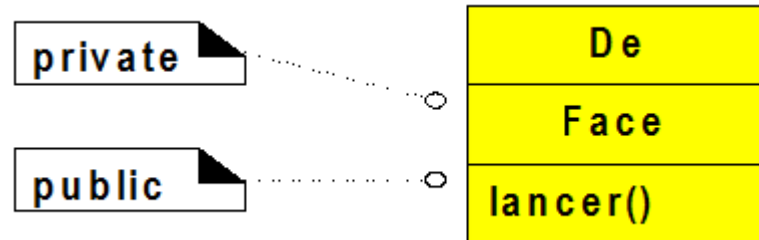
Les objets encapsulent leurs données

- Les données contenues dans les objets sont privées
- Le monde extérieur ne peut les récupérer ou les modifier

Les méthodes sont (en principe) publiques

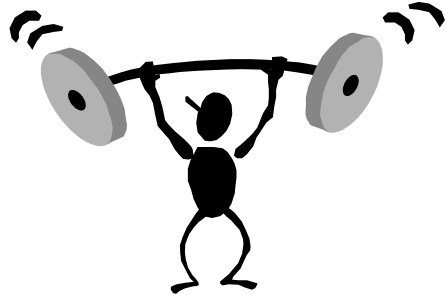
- Le monde extérieur peut envoyer des messages pour les invoquer

Avantages ?

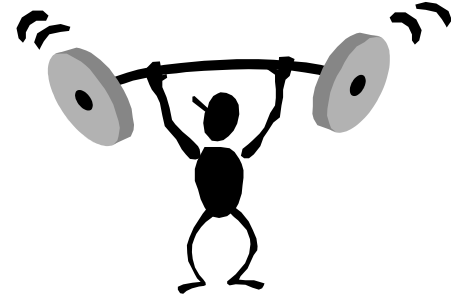


Exercice

Utilisez un outil de développement Java pour définir et tester un objet Gobelet chargé de lancer et de connaître les objets De qu'il contient (voir le code dans les pages suivantes)

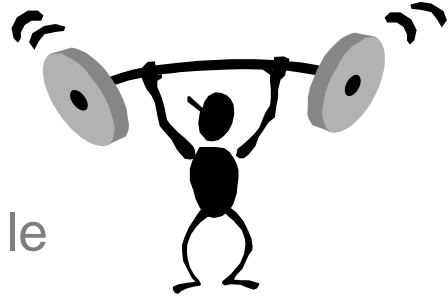


Classe Gobelet — Code source



```
public class Gobelet {  
    private De de1 = new De();    // attribut  
    private De de2 = new De();    // attribut  
  
    public int getFace() { // méthode  
        return de1.getFace() + de2.getFace(); }  
    public void lancer() {      // méthode  
        de1.lancer();  
        de2.lancer(); }  
}
```

Test — Code source



Modifier la classe *Main* précédente pour lancer 10 fois le gobelet au lieu des dés.

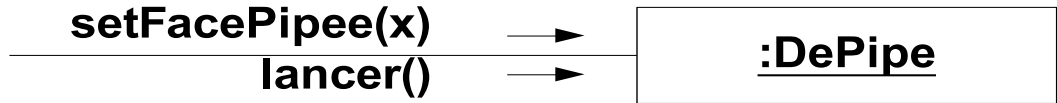
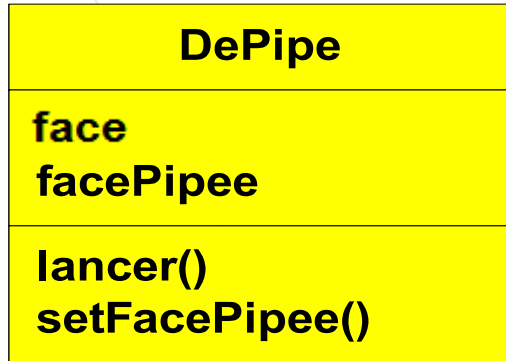
```
...  
    public static void main( String[] args ) {  
        Gobelet g1 = new Gobelet();  
        for (int i = 0; i < 10; i++) {  
            g1.lancer();  
            int value = g1.getValeurFace();  
            System.out.println( value );  
        }  
    } ...
```

Créer une classe de tests unitaires sur gobelet (Optionnel)

Extension de la classe De

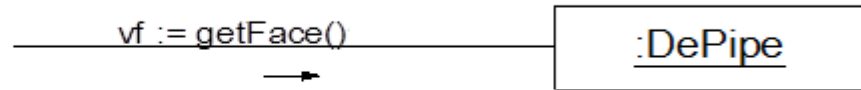
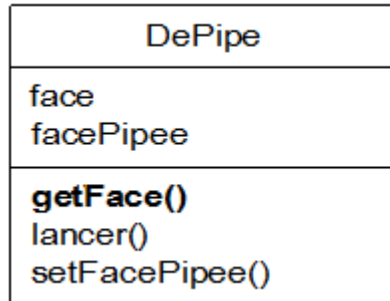
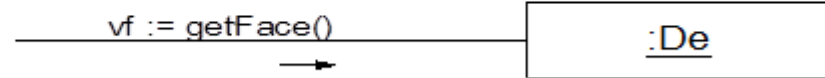
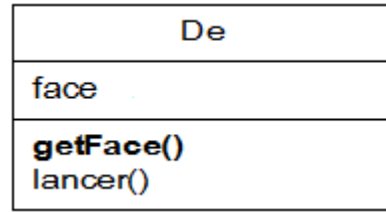
Les objets (composants logiciels) peuvent présenter des variations sur un même thème

Imaginez une nouvelle sorte de dé, un dé pipé, capable de se souvenir d'un côté chargé et fixez la valeur de chaque roulement comme étant égale à la face pipée



De et DePipe : parties communes

Considérez que tous les objets De ont une méthode `getValeurFace` retournant le contenu de `valeurFace`



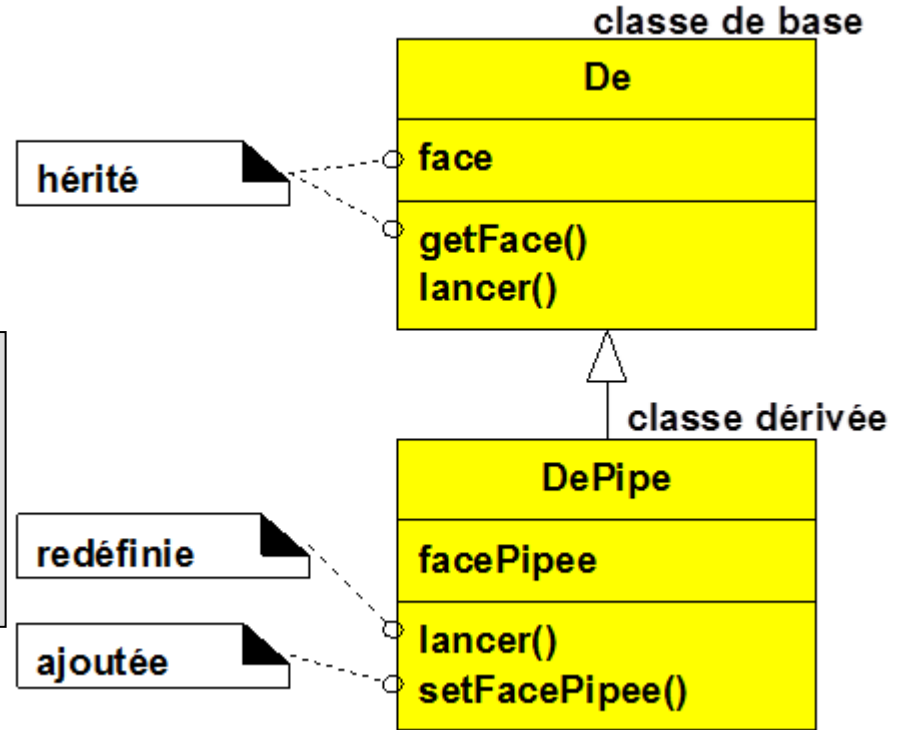
Principe fondamental 2/3: Héritage

L'héritage :

- organisation des classes en une hiérarchie de généralisation-spécialisation
- une sous-classe hérite de toutes les caractéristiques de sa super-classe

```
public class DePipe extends De {  
    private int facePipee;  
    // ...}
```

- Une sous-classe peut redéfinir une méthode héritée



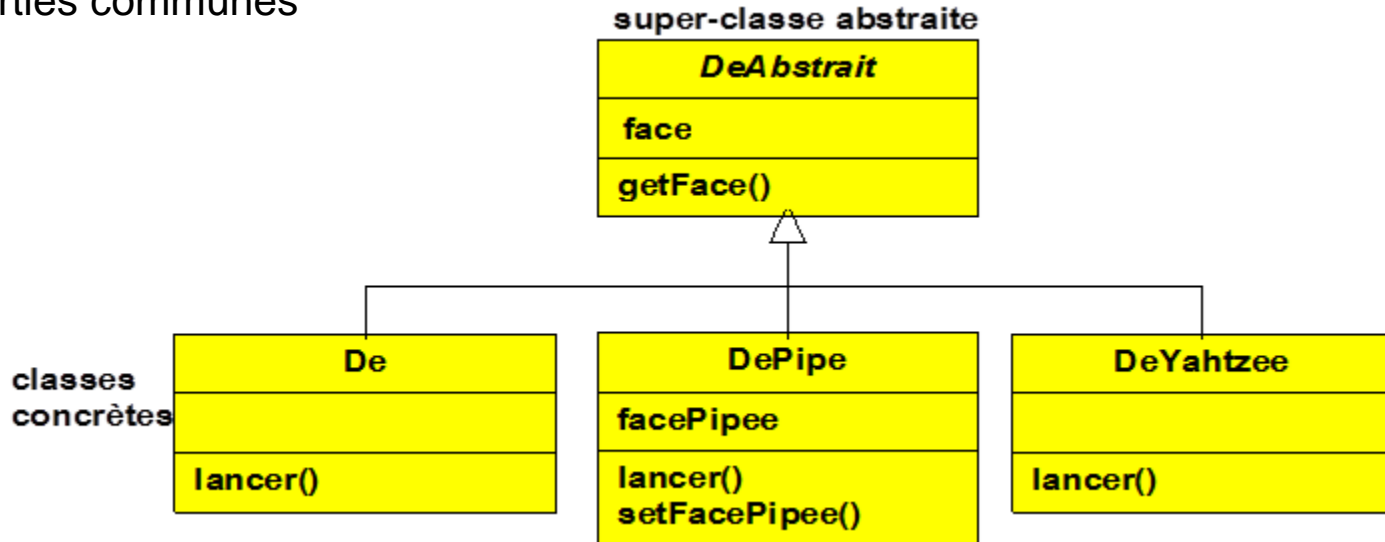
Classes abstraites et concrètes

Classe abstraite

- Ne peut être instanciée
- Super-classe permettant de définir les parties communes

Classe concrète

- Peut être instanciée



Principe fondamental 3/3: Polymorphisme

Le polymorphisme signifie :

- que le même message peut être interprété de différentes façons, selon le récepteur
- que des méthodes portant le même nom peuvent être définies dans différentes classes

Comment savoir quelle méthode sera exécutée lors de l'envoi d'un message ?

Avantages ?

De
face
getFace() lancer()

DePipe
facePipee
setFacePipee(int pipee) lancer()

Terminologie (1/3)

Attribut — information ou état associé à un objet

Classe — définition des caractéristiques et du comportement d'une famille d'objets. « Usine d'instanciation d'objets ».

Classe abstraite — super-classe incomplète définissant les parties communes. Non instanciée.

Classe concrète — classe complète. Classe décrivant un concept de façon complète. Classe destinée à être instanciée.

Agrégation (contenance) — construction d'un composant à partir d'autres composants. Agrégat: objet contenant d'autres objets.

Terminologie (2/3)

Héritage — dans les sous-classes, acquisition automatique des définitions d'attributs et de comportement de la super-classe

Instance — objet créé par une classe

Instanciation — opération de création d'une instance

Message — requête envoyée à un objet pour invoquer une méthode. Un message s'apparente à un appel de fonction, mais il s'en distingue en ceci qu'un objet ne répond qu'à un ensemble prédéterminé de messages définis par sa classe

Méthode — implémentation d'un message. L'envoi d'un message invoque une méthode

Terminologie (3/3)

Objet — composant logiciel. Instance d'une classe. Structure regroupant les attributs et le comportement

Polymorphisme — objets de différentes classes répondant au même message. La réponse de chaque objet risque de varier en fonction de sa classe

Sous-classe — classe définie en termes de super-classe à l'aide de l'héritage

Spécialisation — acte de définition d'une classe comme raffinement d'une autre classe

Super-classe — classe servant de base à l'héritage dans une hiérarchie

Java et la Conception Objet

Architecture de Java, outils et bibliothèques standards

Présentation

Architecture Java

- JVM, JDK, JIT, etc.

Bytecode et fichiers de classes

Environnement et installation

Documentation

Java est compilé ET interprété

1. Une classe est définie dans un fichier « .java » qui porte son nom

1

Code source Java
fichiers *.java

« Write once, run everywhere ! » *

Si c'est une application :
- **java.exe** (Sun)

3. Elle est exécutée (ou interprétée) par une JVM spécifique à chaque plateforme

2

Compilateur Java

- **javac.exe** (Sun)

2. Elle est compilée en un fichier de bytecode « .class »

Java bytecode
fichiers *.class

3

Machine virtuelle Java (Interpréteur)
Win32

Machine virtuelle Java (Interpréteur)
Solaris

Machine virtuelle Java du navigateur Web

Liaison dynamique

Les applications Java se composent de nombreux fichiers .class — et non d'un seul .exe

Les définitions de classes sont lues à partir des fichiers et liées à l'exécution

- On parle de « chargement des classes »
- Fonction du chargeur de classes (Class Loader) de la JVM

Pourquoi ?

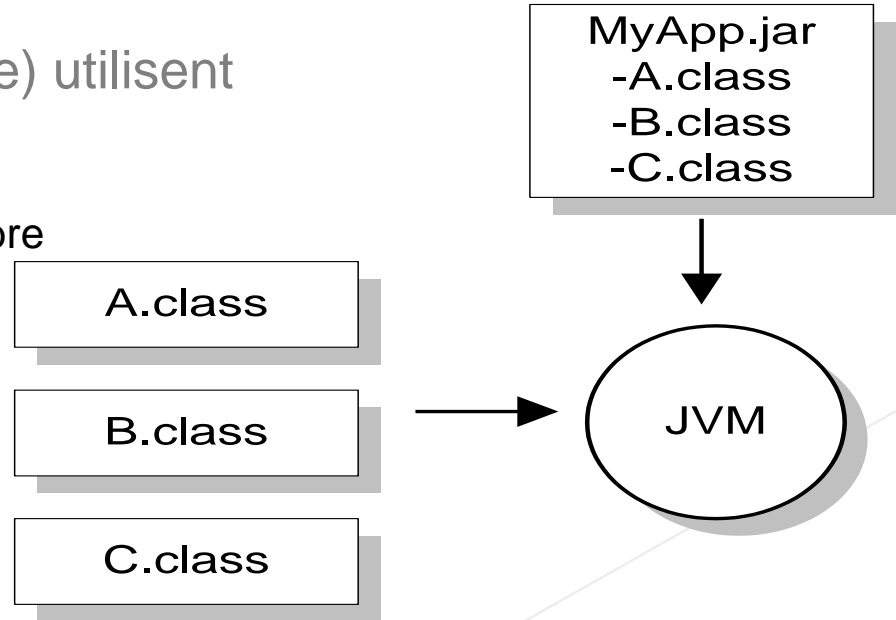
- Plusieurs applications peuvent partager les définitions de classes
 - Semblable aux « .dll » sur Win32
 - Moins de classes à déployer
- De nouvelles classes peuvent être définies ou modifiées à l'exécution

Fichiers .Class et .JAR

Les fichiers de bytecode peuvent se trouver dans des fichiers *.CLASS indépendants ou réunis dans un fichier *.JAR (sorte de fichier ZIP)

Les fichiers JAR (Java ARchive) utilisent
le format de compactage zip

- Utile pour déployer un grand nombre de classes et de fichiers associés.

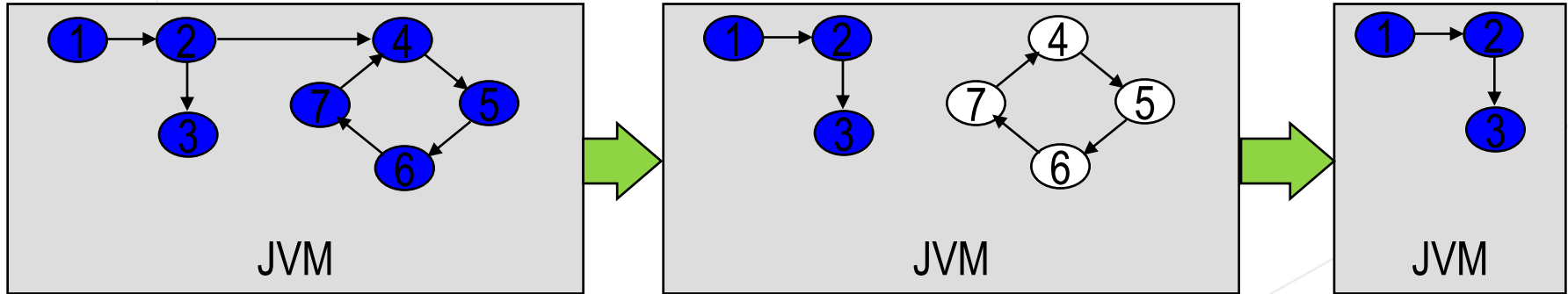


Ramasse-miettes (Garbage Collector)

Les programmeurs Java ne gèrent pas directement la mémoire

Le ramasse-miettes (compris dans la JVM) identifie les objets qui ne sont plus accessibles

- Supprime l'objet en question
- Remet la mémoire à disposition



Par « Java », on entend...

Les outils de déploiement : Java Runtime Environment (JRE)

- Un ensemble standard de bibliothèques disponibles sur toutes les principales plates-formes
 - IHM, accès aux fichiers, ...
- Une JVM

Les outils de développement : Java Software Development Kit (Java SDK)

- Le JRE
- Un ensemble d'outils de développement (compilateur, ...)

La bibliothèque Java

Java comporte une bibliothèque de classes standard prédéfinies

API de base Java 2 :

- Des milliers de classes
- Des structures de données
- Une interface utilisateur graphique
- Des outils de mise en réseau
- ...

java.lang

java.util

java.awt

javax.swing

java.io

java.net

...

Les JVM

Elles transforment le bytecode en code machine

- Suivant les implémentations, elles interprètent le bytecode et appellent des méthodes natives de l'OS, ou elles compilent le bytecode en code natif durant l'exécution.

C'est une spécification que tout le monde peut implémenter:

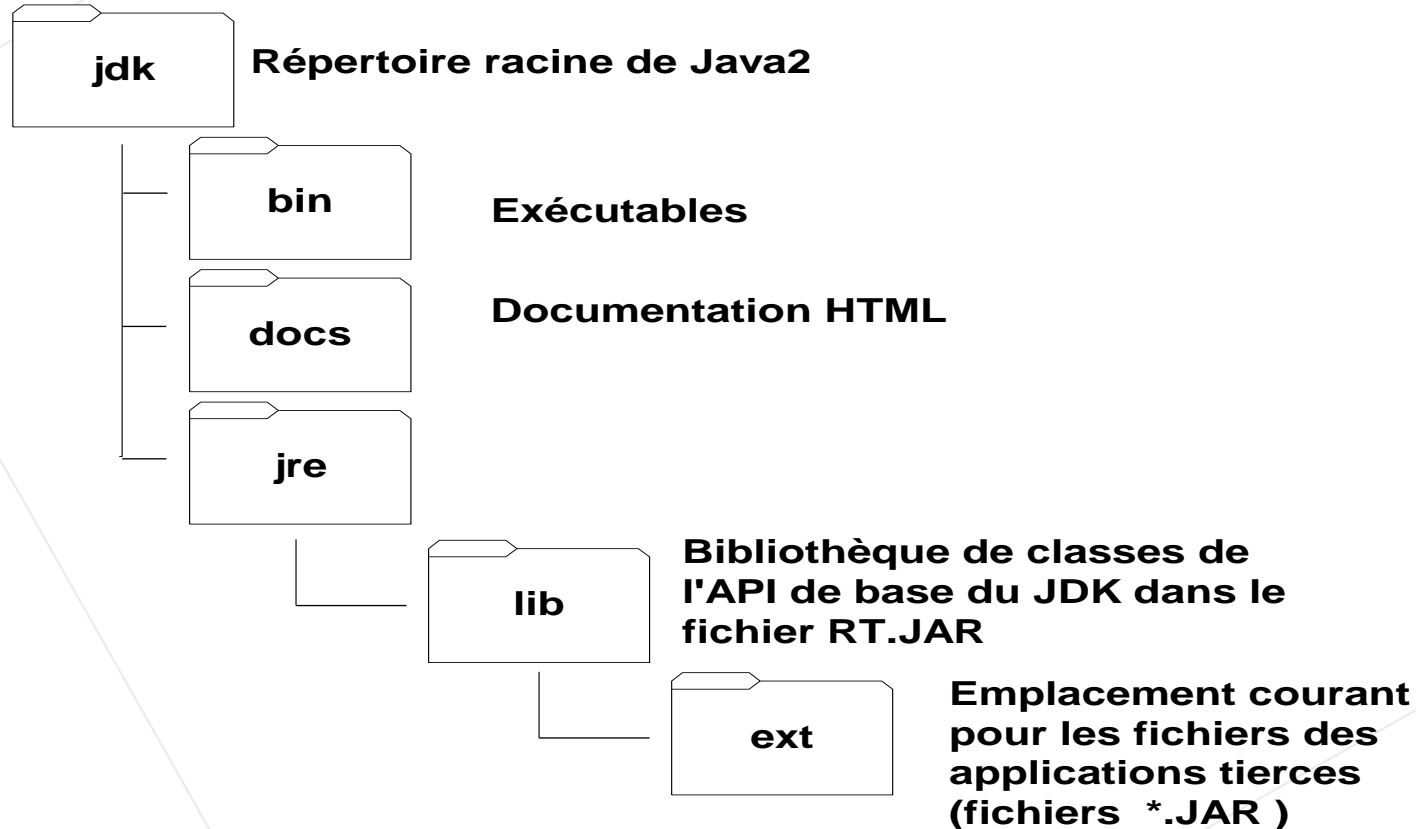
- Sun (HotSpot) est la plus populaire
- IBM en propose également
- Les fournisseurs de serveurs d'application (IBM, BEA,...) créent souvent leur propre JVM sur mesure, pour rendre leur serveurs encore plus performants.
- ...

Outils du SDK d'Oracle (Sun)

Le SDK (Software Development Kit) comprend:

- Compilateur - javac
- Interpréteur (MV) - java
- Applet Viewer - appletviewer
- Débogueur - jdb
- Et tous les autres: jar, javadoc...
- Documentation HTML complète en ligne
- Bibliothèque de classes de base

Répertoires du SDK Java (JDK)



Installation de Java - Variables d'environnement (1/2)

Créez/modifiez la variable PATH pour y intégrer le répertoire contenant les outils Java (java, javac, jar...)

- Win32: `PATH=%PATH%;C:\jdk\bin`
- UNIX: `export PATH=$PATH:/jdk/bin`

Installation de Java - Variables d'environnement (2/2)

La variable d'environnement CLASSPATH indique à Java (javac, java,...) où trouver les fichier .class ou .jar

- Win32: SET CLASSPATH=c:\MyJavaApp1;c:\MyJavaApp2
- UNIX: export CLASSPATH=/MyJavaApp1:/MyJavaApp2

Les fichiers .jar doivent être indiqués explicitement:

- Indiquer le répertoire où ils se trouvent ne suffit pas
- SET CLASSPATH=C:\MyJavaApp\MyJarFile.jar

Vous pouvez inclure le répertoire en cours (".")

- Convention courante
- Attention : peut provoquer un comportement incohérent

La JVM recherchera les classes automatiquement dans le répertoire
jdk\jre\lib\ext

Exécution d'une application

Invoker la JVM pour un fichier *.CLASS (qui peut se trouver à l'intérieur d'un fichier *.JAR)

- > java NomDeClasse
- Ne pas inclure l'extension .class
- c:\> java reservations.TestDe
- Dans cet exemple, la JVM tente d'invoquer la méthode main() dans la classe TestDe du package reservations
- La JVM recherche les répertoires CLASSPATH et \jre\ext pour la classe TestDe

Documentation du JDK

Placée en racine à :

- <jdkDir>\docs\index.html

Où accessible sur :

- <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>

Contient :

- La documentation des bibliothèques de l'API de base du JDK
- Les spécifications du langage, les fonctionnalités, les bugs, ...

The screenshot shows the Java 2 Platform Standard Ed. 5.0 documentation page for the `java.lang.System` class. The page is divided into several sections:

- Overview Package Class Use Tree Deprecated Index Help**: Navigation links at the top.
- Class System**: The class name and its inheritance hierarchy (`java.lang.Object`).
- public final class System**: The class declaration.
- The System class contains several useful class fields and methods. It cannot be instantiated.**: A brief description of the class.
- Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.**: A detailed description of the class's capabilities.
- Since:** JDK1.0
- Field Summary**: A table listing the class's fields.

Field	Summary
<code>static PrintStream err</code>	The "standard" error output stream.
<code>static InputStream in</code>	The "standard" input stream.
<code>static PrintStream out</code>	The "standard" output stream.
- Method Summary**: A table listing the class's methods.

Method	Summary
<code>static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)</code>	Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.
<code>static String clearProperty(String key)</code>	Removes the system property indicated by the specified key.
<code>static long currentTimeMillis()</code>	Returns the current time in milliseconds.
<code>static void exit(int status)</code>	Terminates the currently running Java Virtual Machine.

Java et la Conception Objet

Définition de classes en Java

Présentation

Exploration de la syntaxe, des objets et des idées essentiels de Java

- Syntaxe de base des classes, des attributs et des méthodes
- La méthode `main()`
- Les chaînes (« strings »)
- Les sorties sur console
- Les références d'objets
- Les constructeurs

Conventions de nommage

Les noms de classe commencent par une lettre majuscule

Les noms d'attribut et de méthode commencent par une minuscule

Si un nom comporte plus d'un terme, chaque terme suivant le premier commencera par une majuscule

Type de variable	Source Java
Classe	<code>CompteBancaire</code>
Attribut	<code>tauxDInteret</code>
Méthode	<code>getValeurFace()</code>
Constante	<code>MAX_LANCER</code>
Package	<code>com.valtech.jeudedes</code>

Syntaxe des définitions de classes

On définit une classe par fichier

Le nom de fichier doit correspondre au nom de la classe

Corps de la
définition de
classe entre
accolades

```
public class De {  
    // ...  
}
```

De.java

Syntaxe des attributs

```
public class De {
```

```
    private int valeurFace;
```

```
    private final int MAX_LANCER = 6;  
}
```

type

nom de champ

MAX_LANCER est constant et en lecture seule

Initialisation

Les attributs sont en général privés : seules les méthodes de la classe peuvent y accéder

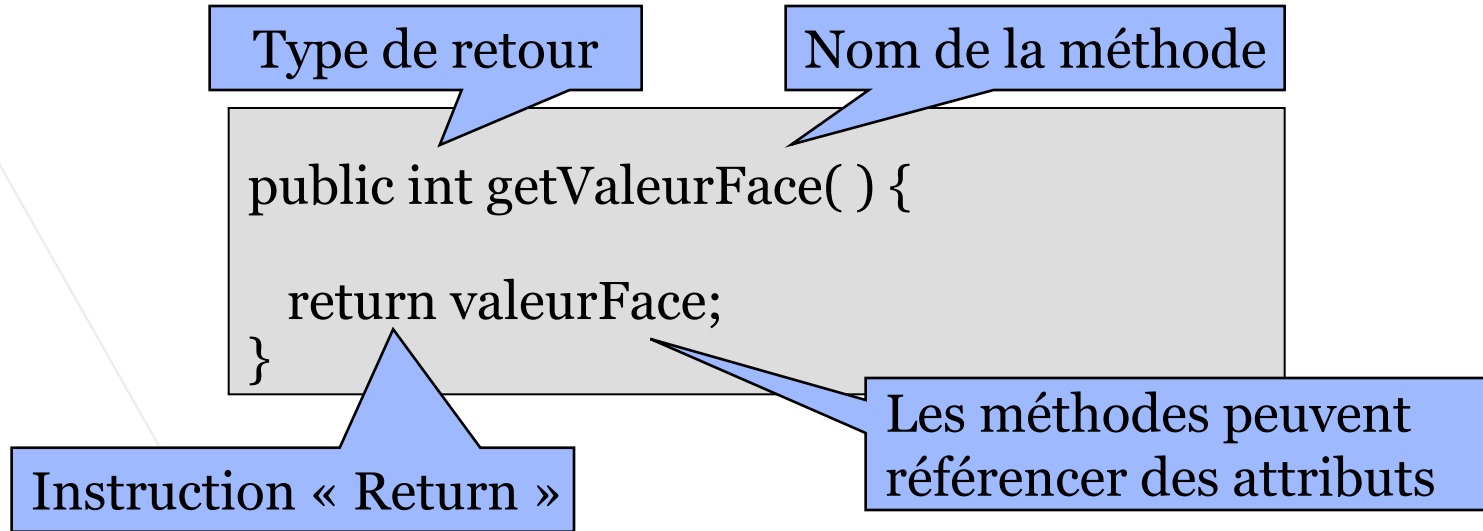
Type	Valeur par défaut
Types numériques	0
char	'\u0000'
boolean	false
Object	null

Méthodes

```
public class De {  
    private int valeurFace;  
    private final int MAX_LANCER = 6;  
    public int getValeurFace() {  
        return valeurFace;  
    }  
    public void setValeurFace( int v ) {  
        valeurFace = v;  
    }  
    public void lancer() {  
        int result;  
        result = (int)(Math.random()*MAX_LANCER) + 1;  
        setValeurFace( result );  
    }  
}
```

Méthode

Syntaxe des méthodes (1/3)



Syntaxe des méthodes (2/3)

Utiliser *void* comme type de retour des méthodes ne retournant rien

Des paramètres peuvent être passés aux méthodes

Les paramètres ne sont visibles que dans le *corps de la méthode*

```
public void setValeurFace( int v )  
{  
    valeurFace = v;  
}
```

Les méthodes peuvent modifier les attributs

Syntaxe des méthodes (3/3)

```
public void lancer() {  
    int result;  
    result = (int)(Math.random()*MAX_LANCER) +1;  
    setValeurFace( result );  
}
```

Les variables locales ne sont visibles que dans le *corps* de la méthode

Les méthodes peuvent invoquer d'autres méthodes

Math.random() retourne une valeur aléatoire comprise entre 0 et 1

Syntaxe des messages

L'appel d'une méthode est désigné sous le terme « envoi de message »

Les messages sont envoyés à un objet en particulier

- Le fait de lancer le dé1 ne modifie pas l'état du dé2

Le format d'un appel de méthode est toujours :

```
public void doIt() {  
    De d = new De();  
    d.lancer();  
    int i = d.getValeurFace();  
    d.setValeurFace( 3 );  
}
```

Invocation d'une méthode

Retour d'une valeur

Passage d'un paramètre

Déclaration, création et affectation

```
public void doIt( ) {
```

Méthode dans une certaine classe.

```
De d;
```

Définition d'une variable locale. Elle ne *pointe* vers rien.

```
d = new De( );
```

Création et affectation d'une instance.

```
De e = new De( );
```

```
e = null;
```

Définition, création et affectation en une étape.

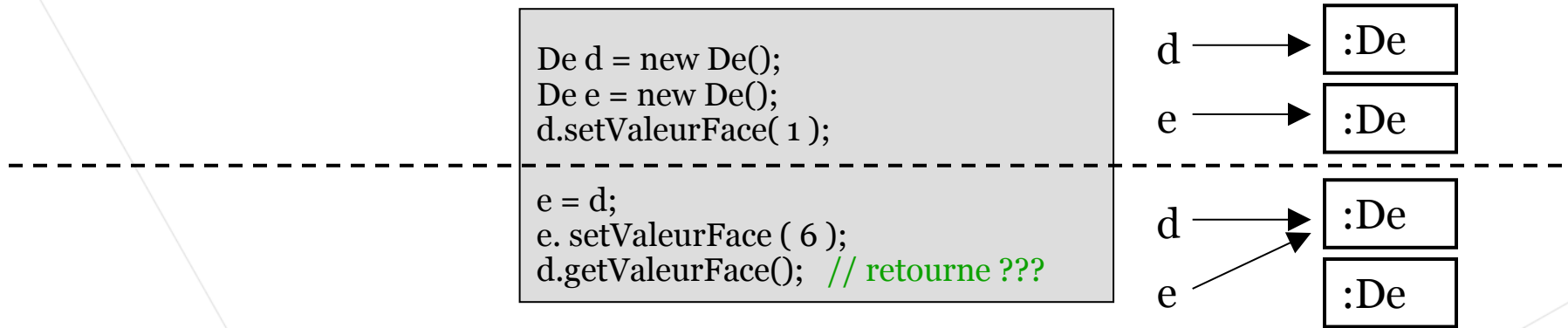
```
}
```

La variable ne pointe plus vers une instance.

Références d'objets et affectation

Une variable référençant un objet ne contient pas l'objet lui-même, mais la référence de celui-ci

L'affectation de l'objet copie la référence (adresse mémoire)



Quand la référence d'un objet est passée en paramètre à une méthode, la référence est copiée (passée par valeur), mais l'objet référencé est toujours l'objet d'origine.

valtech. Toute modification au sein de la méthode modifie l'objet d'origine.

La méthode main()

Première méthode appelée lors du démarrage d'une application

Sert généralement à créer des messages et à les envoyer à des objets

```
public static void main( String[] args ) {  
    De d = new De( );  
    d.lancer( );  
}
```

La classe ayant une méthode main()

Toute classe peut contenir une méthode main()

L'une des pratiques de Java consiste à créer une classe appelée Main contenant la méthode main()

```
public class Main {  
    public static void main( String[] args ) {  
        JeuDeDes j = new JeuDeDes();  
        j.jouer();  
    }  
}
```

Chaînes (« strings »)

Les chaînes sont des instances de la classe String

- `String s = "hello";`
- `// chaîne « littérale » ; style le plus souhaitable`

Les chaînes littérales peuvent également être créées à l'aide de `new`

- Généralement déconseillé, car cela inhibe l'optimisation du compilateur chargé d'éliminer les doublons de chaînes littérales
- `String s = new String("hello");`
- `// autorisé, mais peu souhaitable`

Manipulation des chaînes

Concaténation de chaînes : utiliser l'opérateur +

- "Hello," + "World";

Toutes les primitives peuvent être concaténées avec des chaînes

- int i = 10;
- String s = "i is " + i;

Comparaison de chaînes

```
String s = new String( "hello" );  
// crée 2 chaînes « égales »  
String t = new String( "hello" );
```

```
s == t;           // FAUX!!! Test d'adresse mémoire  
s.equals( t );    // VRAI!!! Test basé sur la valeur
```

Sorties sur console

Avec saut de ligne.

```
void doIt( ) {
```

```
    System.out.println( "Hello World" );
```

Sans saut de ligne.

```
    System.out.print( "Goodbye" );
```

```
    System.out.println( "World" );
```

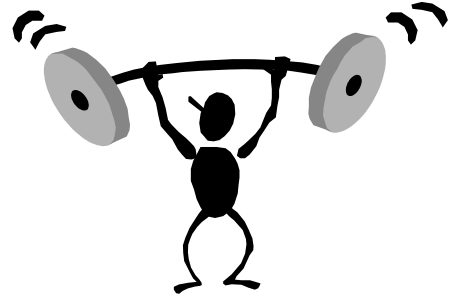
```
    int x = 10;
```

```
    System.out.println( "The value is " + x );
```

```
}
```

Les primitives peuvent être concaténées avec des objets String.

Exercice



Dans la méthode `main()` de la classe `Main` ou dans la classe de tests unitaires :

- Créer un objet `De`
- Afficher la face du dé sans le lancer

```
De d = new De();  
System.out.println( d.getValeurFace() );
```

Quelle est la valeur faciale ?

Constructeurs

Les constructeurs sont des méthodes spéciales qui initialisent une instance

Lors de la création d'une instance, un constructeur est invoqué

```
De d = new De(); // le constructeur va s'exécuter
```

Les constructeurs n'ont pas de type de retour.

```
public class De {  
    public De( ) {  
        lancer( );  
    }  
    // ...  
}
```

Le nom du constructeur DOIT être le même que celui de la classe.

Les constructeurs par défaut

Ne prennent pas de paramètre

- Appelés « par défaut » parce qu'ils attribuent à l'objet des valeurs par défaut raisonnables

Si vous n'écrivez pas explicitement de constructeur, le compilateur en ajoute un par défaut

- (Fondamentalement) ce constructeur ne fait rien
- Son corps est (fondamentalement) celui-ci : {}.

Attention! Si vous spécifiez un constructeur quel qu'il soit, le constructeur par défaut ne sera plus généré



Constructeurs multiples et paramètres des constructeurs

Une classe peut avoir de nombreux constructeurs

Les multiples constructeurs se distinguent les uns des autres par la liste de leurs différents paramètres

```
public class De {  
    public De( ) {  
        lancer( );  
    }  
  
    public De( int valeurInitiale ) {  
        face = valeurInitiale;  
    }  
    // ...  
}
```

Invocation des constructeurs

Un constructeur n'est pas invoqué directement, mais à travers l'opération new

```
De d;
```

```
d = new De( );
```

```
d = new De( 3 );
```

Invocation du constructeur sans paramètre.

Invocation du constructeur à un argument pour initialiser la valeurFace.

Variables ambiguës

Ce code manque de clarté pour le compilateur :

```
public class De {  
    private int face;  
  
    public De( int face )    {  
        face = face;  
    }  
    // ...
```

« valeurFace » est-il le paramètre ou l'attribut ?

Mais si vous voulez écrire un code équivalent...

Utilisation de « this » (1/2)

La variable spéciale « this » (terme réservé) clarifie la signification du code

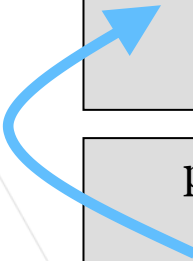
« this » renvoie à l'objet courant. *this.face* désigne donc l'attribut *face* de l'objet courant

```
public class De {  
    private int face;  
  
    public De( int face)  {  
        this.face = face;  
    }  
    // ...
```

Voici donc le style Java à utiliser pour les constructeurs.

Utilisation de « this » (2/2)

this renvoie à l'objet sur lequel la méthode a été invoquée

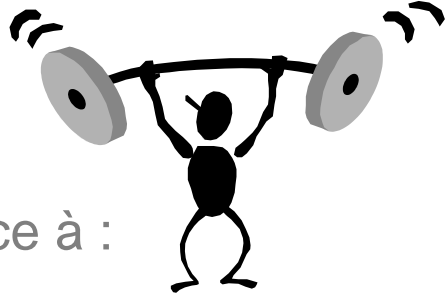


```
De d1 = new De( );  
d1.lancer( );
```

```
public void lancer( ) {  
    this.setFace( val); // possible, mais inusité  
    setFace( val );    // équivalent ; idiomatique  
}
```

Le compilateur ajoute implicitement « *this* » aux appels de méthode lorsque le récepteur n'est pas spécifié

Exercice : ajout de constructeurs



Ajoutez des constructeurs à la classe `De` initialisant face à :

- un entier au hasard compris entre 1 et 6
- une valeur passée en tant qu'entier au constructeur

Ajouter des tests unitaires sur les constructeurs dans la classe *TestDe*

Java et la Conception Objet

Autres éléments de syntaxe Java

Présentation

Jetons un coup d'œil à d'autres éléments de la syntaxe Java

- Les types de base (ou primitives)
- Les littéraux
- Les variables
- Les opérateurs mathématiques
- Les commentaires
- Les blocs

Types de base (primitives)

Type	Taille	Valeurs
boolean	1 bit	« true » ou « false »
char	16 bits	Unicode \u0000 à \uFFFF
byte	8 bits	-128 à 127
short	16 bits	-32768 à 32767
int	32 bits	-2147483648 à 2147483647
long	64 bits	-9223372036854775808 à 9223372036854775807
float	32 bits	+/- 3.402E+38 à +/-1.402E-45
double	64 bits	+/- 1.798E+308 à +/- 4.941E-324

Les primitives ne sont pas des objets

Ce sont de simples valeurs

- Pour des raisons de performance

Une variable de type primitive ne contient pas une référence vers une primitive, mais directement la valeur de celle-ci

Quand une variable de type primitive est passée en paramètre à une méthode, elle est copiée (passée par valeur)

- Une modification du paramètre au sein de la méthode laisse la variable d'origine intacte
- (A la différence des objets, qui sont passés par référence)

Littéraux

Type des littéraux numériques

- int pour les littéraux sans décimale : 3, 121, 3420
 - Pour créer un entier littéral de type long, faites-le suivre d'un « l » ou d'un « L » (3l, 121L, 3420l)
- double pour les littéraux à décimale: 3.0, 1.21, 34.2
 - Postfixer par F ou f pour spécifier un float

Types des littéraux à caractère

- Ce sont des caractères Unicode (16 bits)
- Des antislash (\) précèdent les littéraux à caractères non imprimables ('\n', '\t',...)

Déclaration de variables

Il faut déclarer les variables avant de pouvoir les utiliser

```
int i;  
double d;  
De monDe;
```

```
int j, k, l;
```

Primitive: la variable contient directement la valeur

Objet: la variable contient la référence de l'objet

Plusieurs variables peuvent être déclarées sur la même ligne

```
int i = 10;  
De monDe = new De();
```

À moins que leur valeur par défaut ne convienne

Elles doivent être initialisées

Conversion de primitives Java

```
int i = 10;  
double d = 3.42;
```

```
d = i;  
i = d;
```

OK. Fait passer un entier en double.

Erreur ! Pourquoi ?

```
i = ( int ) d;
```

OK. Les conversions qui tronquent les primitives doivent utiliser des « casts » explicites.

```
byte b = 34L;
```

Erreur ! Pourquoi ?

```
byte b = (byte) 34L;
```

Attention aux limites des primitives.
La valeur pourrait être abhérente.

Opérateurs mathématiques

<i>opérateur</i>	<i>but</i>	<i>exemple</i>	<i>x devient</i>
=	Affectation	x = 10	10
*	Multiplication	x = 10 * 3	30
/	Division	x = 10 / 3	3
%	Modulo	x = 10 % 3	1
+	Addition	x = 10 + 3	13
-	Soustraction	x = 10 - 3	7

Incrémentation/décrémentation

- « ++ » incrémente
- « -- » décrémente

```
int i = 10;
```

```
i++;
```

```
i--;
```

i vaut 11

i vaut 10

Opérateurs d'incrémentation

« ++ » et « -- » peuvent être placés en préfixe ou en suffixe

- Les opérateurs en préfixe s'exécutent avant tout autre opérateur dans l'instruction
- Les opérateurs en suffixe s'exécutent après l'exécution de tout autre opérateur dans l'instruction.

```
int i = 10;  
int j = 0;
```

```
j = ++i;
```

j égale 11, i égale 11

```
i = 10;
```

```
j = i++;
```

j égale 10, i égale 11

Variables « final » et commentaires

Les variables « final » sont constantes et en lecture seule

```
final int MAX = 10;  
int i = MAX;  
MAX = 20;
```

OK. Lecture à partir d'une variable « final ».

Erreur ! MAX est en lecture seule.

Commentaires et la *Javadoc*

- est un outil permettant, de produire une documentation très complète de votre code en une page Web.
- Il est visible dans les *IDE*

```
// commentaire sur une ligne
```

```
/* commentaire sur plusieurs  
lignes */
```

```
/**
```

```
 * @author anne-lise.dubas
```

```
 * @category calcul de la face du dé (6 faces possibles)
```

```
 */
```

```
public void lancer() {
```

Blocs

{ définit un bloc de code. Les blocs regroupent les instructions et limitent la portée d'une variable

```
{  
    {  
        int x = 0;  
        x = 4;  
        {  
            int y = 0;  
            y ++;  
        }  
        y = 10;  
        x = 7;  
    }  
    x = 20;  
}
```

ERREUR ;
y est hors de portée.

ERREUR ; x est hors de portée.

Portée: les variables
déclarées à l'intérieur
d'un bloc n'existent
que dans ce bloc

Java et la Conception Objet

Structures de contrôle Java

Présentation

Nous allons explorer les diverses structures de contrôle de Java

- Instructions « if »
- Opérateurs booléens
- Utilisation de « == » pour la comparaison des identités
- Instructions « switch »
- Boucles « while »
- Boucles « for »

Structures conditionnelles

Structures conditionnelles sur une seule ligne

```
if( conditionalExpression )  
    statement;
```

Exemple :

```
public void doIt( int x ) {  
    if ( x < 10 )  
        x++; // structure conditionnelle  
        // l'instruction suivante est non conditionnelle  
    y++;  
}
```

Programmeurs C, attention !

Seules les expressions booléennes sont autorisées dans les structures conditionnelles

```
int i = 10;  
if ( i ) //ERREUR  
    x++;
```

Structures conditionnelles à plusieurs lignes

On utilise des blocs pour grouper plusieurs instructions en une seule structure conditionnelle

```
public void doIt( int x ) {  
    if ( x < 10 )  
    {  
        x++;  
        z++;  
    }  
}
```

Ces deux instructions sont exécutées si $x < 10$.

Instructions « else if »

```
if ( x < 10 ) {  
    // code conditionnel  
}  
else if ( x < 20 ) {  
    // première possibilité  
}  
else {  
    // dernière possibilité  
}
```

Opérateurs conditionnels

Opérateur	Description
<, <=, >, >=	Comparaison numérique
==, !=	Relationnel (égalité, inégalité)
!	« Non » logique
&&,	Opérateurs et/ou

Exemple

```
public void doIt( int i ) {  
    if ( ( i < 0 ) || ! ( i < 10 ) ) x++;  
}
```

Le test ==

« == » compare les adresses en mémoire, pas les valeurs

```
De d = new De( 5 );  
De e = new De( 5 );
```

Les *objets* *d* et *e* ont chacun une valeurFace de 5.

```
if ( e == d )  
    x++;
```

Toujours « false »
== n'examine pas les valeurs des variables d'instance.

```
e = d;
```

```
if ( e == d )  
    y++;
```

Renvoie « true »
d et *e* référencent le même objet.

L'instruction « switch »

Applicable aux types int, short, byte, char , au énumération (Java 5) et aux String (Java 7)

```
switch( j ) {  
    case 1:  
        x++;  
        break;  
    case 2: case 3: case 4:  
        --x;  
        y = 5;  
        break;  
    default:  
        x = 0;  
        y = y + 9;  
}
```

L'instruction « switch » avec enum

Enum permet de typer la variable et de contrôler la liste de ses valeurs possibles (depuis le SDK 1.5)

```
public enum Saison{ HIVERS, PRINTEMPS, ETE, AUTOMNE };

public static void afficher( Saison saison) {
    switch(saison) {
        case HIVERS : System.out.println( "Hivers" );
        break;
        Default : System.out.println(saison.toString());
        break;
    }
}

public static void main( String[] args ) {
    afficher( Saison.HIVERS ); }
```

Les boucles « while »

```
while( boolean expression )  
{  
    // instructions exécutées de façon conditionnelle  
}
```

Exemple :

```
int i = 0;  
while( i < 10 ) {  
    System.out.println( "i is " + i );  
    i++;  
}
```


Les instructions « for »

Initialisation

Expression booléenne

Incrément

```
for( int i = 0; i < 10; i++ )  
{  
    System.out.println( "i is " + i );  
}
```

Continue tant que l'expression booléenne est vraie
i ne peut être référencé qu'à l'intérieur du bloc for
; sépare chaque partie de la boucle for

Java et la Conception Objet

**Collaboration entre objets –
relations, agrégation
(« *containment* »)**

Présentation

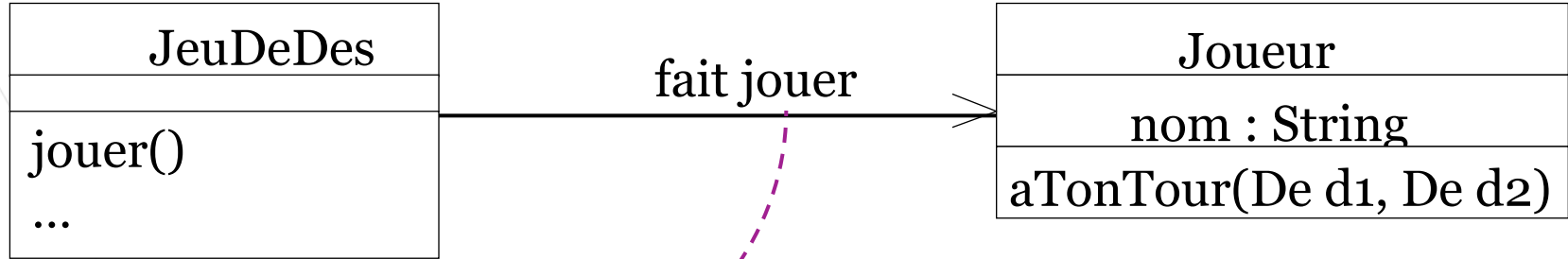
Les objets effectuent les tâches en collaborant avec d'autres objets (et en leur déléguant leurs attributions)

On dit d'un objet qui entretient une relation permanente avec un autre objet qu'il « contient » cet objet

On utilise des accesseurs (« getters ») et des mutateurs (« setters ») pour exposer les attributs d'un objet à d'autres objets

Agrégation d'objets

Un objet peut contenir d'autres objets



```
public class JeuDeDes{  
    private Joueur [] joueurs = new Joueur[5];  
    //...  
}
```

Collaboration avec des objets contenus

```
public class JeuDeDes{  
  
    private Joueur [] joueurs = new Joueur[5];  
    private De d1, d2;  
  
    public void jouer()  
    { ...  
        joueurs[i].aTonTour(d1, d2);  
    }  
}
```

Méthodes d'accès et de mutation

Les méthodes d'accès, ou accesseurs, récupèrent la valeur des attributs (Ce sont des méthodes get)

Les méthodes de mutation, ou mutateurs, modifient la valeur des attributs (Ce sont des méthodes set)

```
class De {  
    private int face;  
  
    public int getFace( ){  
        return face;  
    }  
  
    public void setFace( int v ) {  
        face = v;  
    }  
    //...
```

Accesseur

Mutateur

Exemple

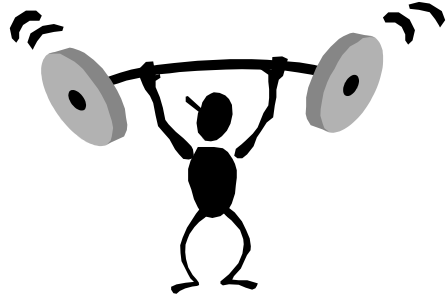
Les variables d'instance ont, en général, des méthodes d'accès et de mutation correspondantes

On utilise (en principe) des méthodes d'accès et de mutation pour accéder aux données d'un objet

- En quoi est-ce une bonne habitude ?

```
public class Main{  
    public static void main( String[] args )  {  
        De d = new De();  
        d.setValeurFace( -100 );  
        int i = d.getValeurFace( );  
        ...  
    }  
}
```

Exercice collaboration d'objets (1/2)



Ajoutez une classe Joueur au projet JeuDeDes

Utilisez comme guide le diagramme de la page suivante

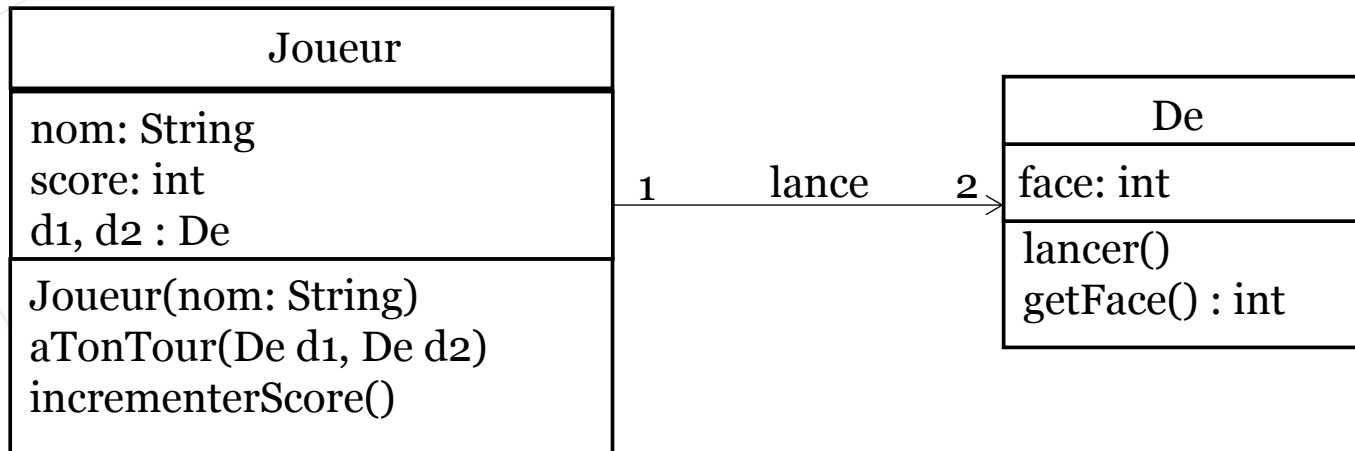
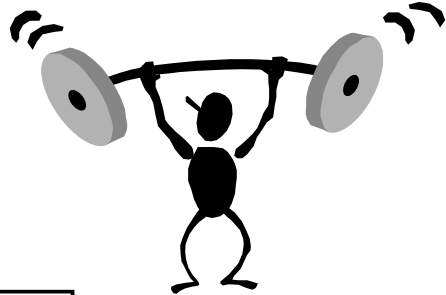
Les attributs du Joueur sont un nom, un score

- Pour l'instant, assignez au Joueur les attributs d1 et d2

Le Joueur dispose de trois méthodes :

- aTonTour(De d1, De d2),
 - Lance les deux dés
 - Incrémente le score (en appelant incrementerScore()) si la somme des dés est supérieure à six
- incrementerScore() incrémente l'attribut score
- et un constructeur
 - Passez une chaîne comme paramètre et utilisez-la pour attribuer le nom
 - Instanciez les objets d1 et d2
 - Initialisez l'attribut score à zéro

Exercice collaboration d'objets (2/2)



Créer une classe de tests unitaires *TestJoueur* pour valider votre code

Java et la Conception Objet

Tableaux d'objets et de primitives

Tableaux (« arrays »)

Ils détiennent une collection de taille fixe

Ils détiennent des objets ou des primitives

Les tableaux en Java sont des objets — des instances d'une classe spéciale de tableaux

- Les tableaux doivent être créés à l'aide de new.

Création de tableaux

Déclarations de tableaux

- Tableau de références sur des primitives
 - `float[] nombres;`
- Tableau de références sur des objets
 - `De[] des;`

Créations de tableaux

- Utiliser l'opérateur `new` et spécifier la taille du tableau entre `[]`
`float[] nombres = new float[4];`

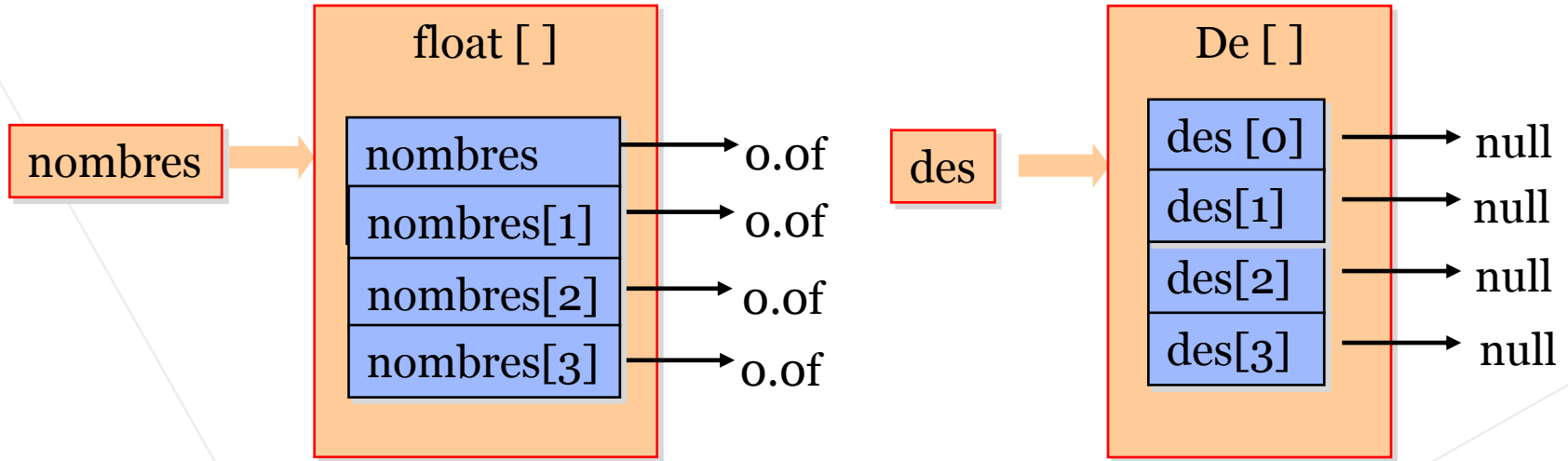
`De[] des = new De[4];`

Nombre d'éléments
contenus dans le tableau.

Création de l'instance
de tableau.

Initialisation de tableaux

Après la création, les éléments du tableau doivent être initialisés à des valeurs par défaut



Accès aux éléments d'un tableau

« [] » permet d'accéder aux éléments d'un tableau et de les modifier

Les indices des tableaux commencent à zéro

Les indices des tableaux sont contrôlés

```
float[ ] nombres = new float<[ 4 ];
```

```
nombres[ 0 ] = 10;
```

```
nombres[ 1 ] = 2;
```

```
nombres[ 2 ] = 70;
```

```
nombres[ 3 ] = 1;
```

```
float i = nombres[ 1 ];
```

```
nombres[ 4 ] = 7;
```

ERREUR D'EXECUTION !

Une affectation hors des bornes met fin au programme.

Longueur des tableaux

length est un attribut public indiquant le nombre d'éléments du tableau

L'attribut length est en lecture seule

```
void foo( float[] f ) {  
    for( int i = 0; i < f.length; i++ )  
        f[ i ] = i;  
}
```

Initialisation de tableaux d'objets

Créer chaque élément séparément

```
// créer un tableau vide
```

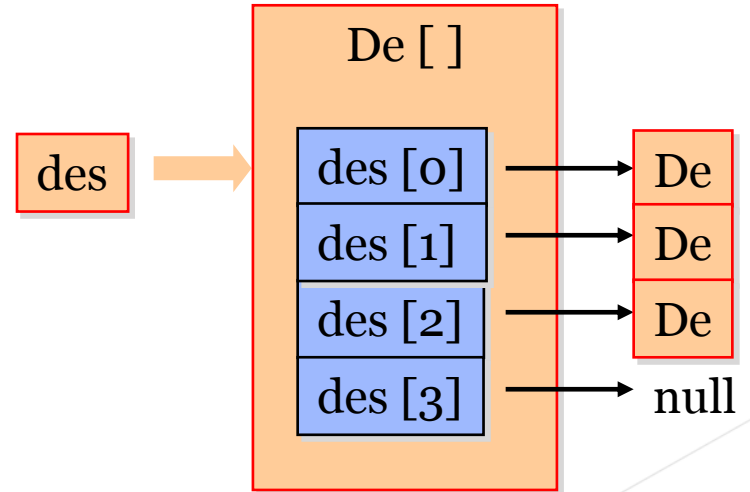
```
De[ ] des = new De[ 4 ];
```

```
// le charger d'objets
```

```
des [ 0 ] = new De( 2 );
```

```
des [ 1 ] = new De( 4 );
```

```
des [ 2 ] = new De( 6 );
```



Initialiseurs de tableaux

Les initialiseurs de tableaux permettent de créer et d'initialiser des tableaux

Placer les valeurs initiales
entre { }.

```
int[ ] nombres = new int[ ] { 2, 4, 6 };  
De[ ] des = new De[ ] { new De( ), new De( ) };
```

L'instruction new peut être implicite

```
De[ ] des = { new De( ), new De( ) };  
int[ ] nombres = { 2, 4, 6 };
```

new [] implicite

Utilisation de tableaux d'objets

Les éléments du tableau sont en principe immédiatement initialisés

```
for ( int i = 0; i < des.length; i++ )  
    des[ i ] = new De( );
```

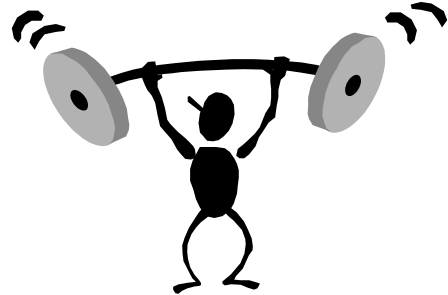
Envoi de messages aux éléments du tableau en utilisant length

```
for ( int i = 0; i < des.length; i++ )  
    des[ i ].lancer( );
```

Envoi de messages aux éléments du tableau en utilisant un Foreach

```
for ( De de : des )  
    de.lancer( );
```

Exercice 1



Modifiez la méthode `main()` afin de :

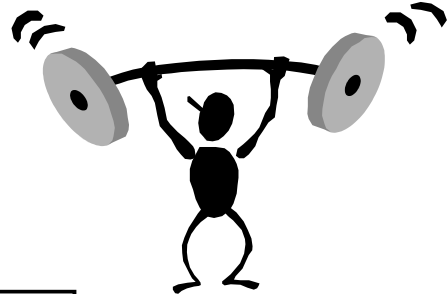
Première partie :

- créer un tableau de Joueurs
- créer cinq instances de « Joueur » et les ajouter au tableau
 - Utiliser « Joueur1 », « Joueur2 », « Joueur3 » (etc.) comme nom des Joueurs
 - `joueurs[i] = new Joueur("Joueur " + (i+1));`
- permettre à chaque joueur de jouer 10 fois

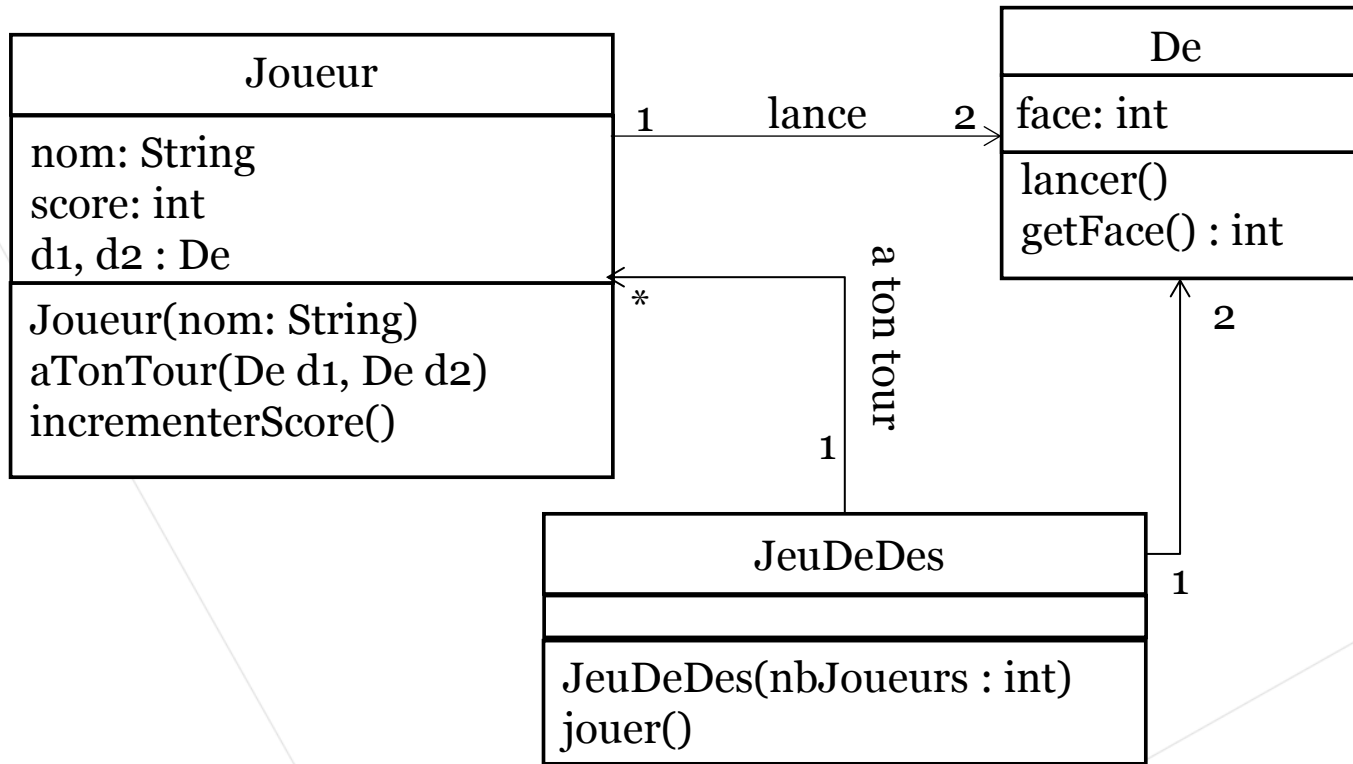
Deuxième partie :

- Rendre compte du ou des vainqueurs de la partie

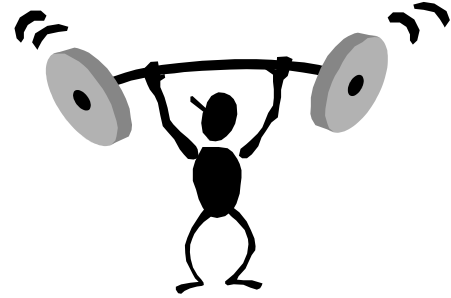
Exercice 2 (1/2)



Ajoutez la classe *JeuDeDes* (voir à la page suivante)



Exercice 2 (2/2)



Déclarez comme attribut de *JeuDeDes* :

- un tableau de Joueurs
- Les 2 dés (Modifier la classe Joueur en conséquence)

C'est le constructeur qui doit créer ce tableau et le remplir d'objets Joueur

- Passez le nombre de Joueurs comme paramètre du constructeur.
- Initialiser les 2 dés.

La méthode *jouer()* permet à chaque joueur de jouer dix fois, puis rend compte du ou des joueurs totalisant le plus grand nombre de victoires

Dans la méthode *main()*:

- créez un objet *JeuDeDes*
- invoquez la méthode *jouer()* sur l'instance

Java et la Conception Objet

Les packages

Présentation

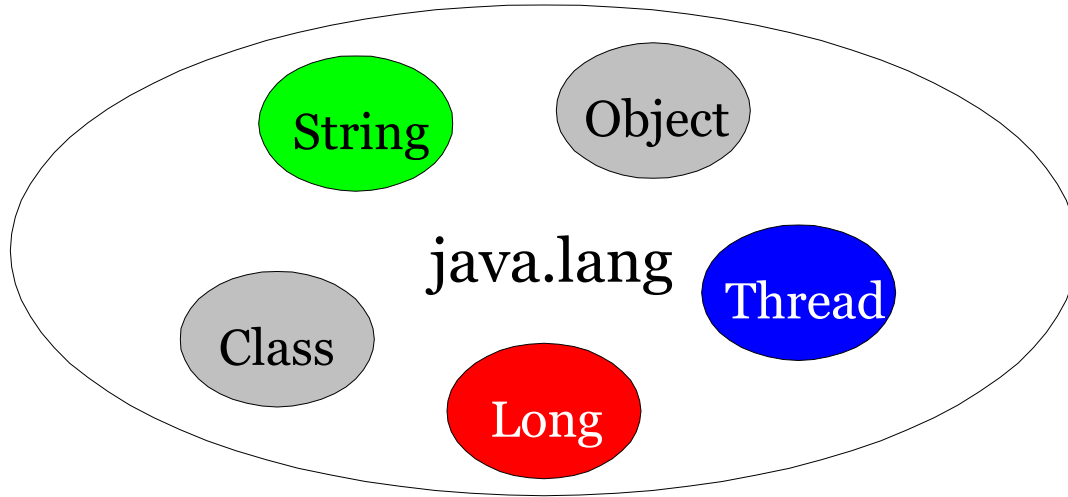
Java fournit le mécanisme de « package » pour organiser et gérer les ensembles de types (classes et interfaces)

Objectifs :

- Définir les packages et leur utilité
- Importer des classes à partir de packages
- Comprendre la relation existant entre les noms de package et les répertoires
- Comprendre de quelle façon Java localise les fichiers .class à l'aide des noms de package et de CLASSPATH

Packages

Un package est une collection de classes et d'interfaces



Pourquoi les packages ?

Ils permettent d'organiser les classes sous forme d'unités plus vastes

- De la même façon que les répertoires organisent les fichiers et les applications

Ils permettent d'éviter les problèmes de conflits entre noms

- Problème qui se généralise avec l'augmentation continue du nombre de classes Java

Ils protègent les classes et les membres à plus haut niveau que classe par classe

- Encapsulation au « niveau du package »

API de base de Java

La bibliothèque Java standard couvre un certain nombre de packages

- `java.lang`
- `java.util`
- `java.awt`
- `java.awt.event`
- `javax.swing`
- `java.io`
- `java.net`
- ...

L'instruction « package »

L'instruction « package » déclare qu'une classe ou une interface appartient à un package

Il doit s'agir de la première ligne sans commentaire du fichier source

```
package java.util;
```

```
public class Date {
```

```
//...
```

```
}
```

L'instruction « package » dit la chose suivante : « cette classe va dans ce package ».

Une seule instruction « package » par fichier .java

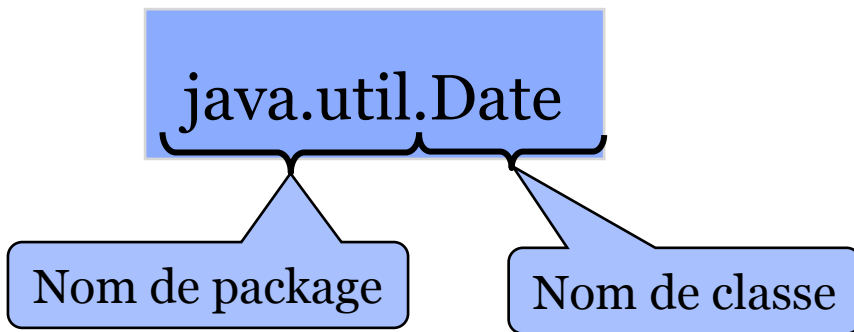
Packages et noms de classe

Pour être parfaitement complet, le nom de classe doit comprendre le nom du package :

- `Date d = new Date;`

est en réalité :

- `java.util.Date d = new java.util.Date();`



Utilisation de classes de différents packages

On peut toujours utiliser le nom de classe complet

- `java.util.Date d = new java.util.Date();`
- ou...

Utilisez l'instruction `import` :
*elle rend les classes
accessibles par leur nom
abrégé.*

```
package monpackage ;

import java.util.Date;
import java.util.LinkedList;

public class MaClasse
{
    Date d = new Date( );
    LinkedList l = new LinkedList();
}
```

Importation de packages entiers

Un astérisque (*) permet d'importer tous les types publics à partir d'un package

```
import java.util.*;  
...  
LinkedList v = new LinkedList( );  
Date d = new Date( );  
HashMap hm = new HashMap( );
```

Ces classes
sont dans
java.util.

import permet seulement aux développeurs d'utiliser les noms de classe abrégés dans le fichier source en cours

- Les bytecodes utilisent toujours les noms de classe complets
- Pas d'impact à l'exécution ; ni en taille, ni en performance
- Différent de #include en C

Import de packages - conflits de noms

Si deux packages ont des classes portant le même nom, on peut les importer tous les deux, mais cela crée une ambiguïté

Par exemple, il y a deux classes nommées « Date » :

- java.util.Date
- java.sql.Date

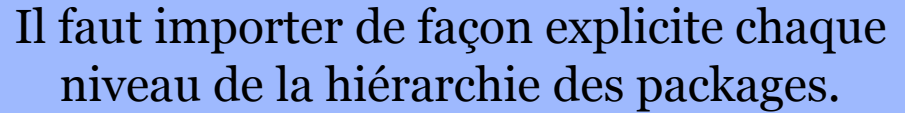
```
import java.util.*;  
import java.sql.*;  
...  
Date d = new Date( );  
// erreur du compilateur — quelle Date ?
```

Solution : utiliser des noms de classe pleinement qualifiés (ou complets)

Import de packages - Sous-packages

L'astérisque (*) ne permet pas d'importer les « sous-packages »

- `import java.awt.*;`
- `import java.awt.event.*;`
- `import java.awt.image.*;`



Il faut importer de façon explicite chaque niveau de la hiérarchie des packages.

Packages automatiquement importés

Certains packages sont automatiquement importés

- java.lang
- Le package actuellement nommé

```
package modele;  
  
import modele.*; // INUTILE  
import java.lang.*; // INUTILE  
  
class JeuDeDes {  
    Joueur[] joueurs; // Joueur est dans le même package  
    ...  
}
```

Packages et répertoires

Il est essentiel de comprendre la relation qui unit les noms de package et de savoir à quel endroit sont stockés les fichiers .class dans la structure des répertoires !

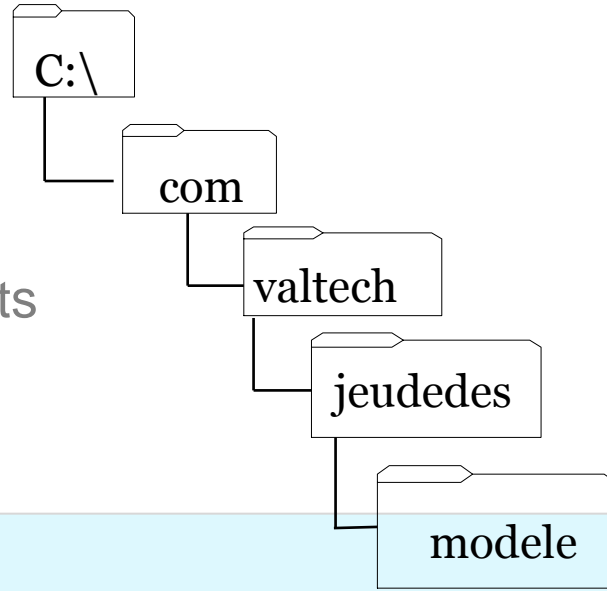
Tous les fichiers .class d'un même package doivent être localisés dans un sous-répertoire correspondant au nom complet du package

Exemple

Ces fichiers source indiquent le nom du package
'com.valtech.jeudedes.modele'

Les fichiers *.class* correspondants doivent donc résider dans un sous-répertoire nommé
'com.valtech.jeudedes.modele'

Avec le compilateur *Javac*, les fichiers sources sont généralement situés dans le même répertoire



JeuDeDes.java

```
package  
com.valtech.jeudedes.modele;
```

```
public class JeuDeDes{ ... }
```

Joueur.java

```
package  
com.valtech.jeudedes.modele;
```

```
class Joueur{ ... }
```

JeuDeDes.class

Joueur.class

Packages, répertoires et CLASSPATH

Il est essentiel de comprendre la variable d'environnement CLASSPATH et sa relation avec les noms de package et la structure des répertoires !

Comme nous l'avons noté plus haut, tous les fichiers .class d'un package doivent être situés dans un sous-répertoire correspondant au nom complet du package

La variable d'environnement CLASSPATH spécifie les répertoires racines à partir desquels se ramifient les sous-répertoires de packages

Exemple

Supposons que notre code réside dans le package nommé « com.valtech.jeudedes.modele ». Supposons que le code est rangé dans les sous-répertoires nommés « c:\jod »

C:\jod\com\valtech\jeudedes\modele

JeuDeDes.java

```
package com.valtech.jeudedes.modele;  
  
public class JeuDeDes{ ... }
```

JeuDeDes.class

Joueur.java

```
package com.valtech.jeudedes.modele;  
  
class Joueur{ ... }
```

Joueur.class

Quel répertoire doit se trouver dans le CLASSPATH ?

Localisation des fichiers .class par Java

CLASSPATH=.;c:\project\java;c:\jod;

Code source :

```
import com.valtech.jeuedes.modele.*;  
...  
JeuDeDes jeu = new JeuDeDes(20,6 );
```

Spécifie trois répertoires.

ou bien...

```
com.valtech.jeuedes.modele.JeuDeDes jeu = new com.valtech.jeuedes.modele.JeuDeDes(20,6);
```

Java recherche JeuDeDes.class de cette façon :



```
.\com\valtech\jeuedes\modele\JeuDeDes.class  
c:\project\java\com\valtech\jeuedes\modele\JeuDeDes.class  
c:\jod\com\valtech\jeuedes\modele\JeuDeDes.class
```

Package non nommé

Si le fichier source n'a pas de déclaration de package, Java place la classe dans un « package non nommé »

Déconseillé

Pourquoi ?

```
// pas d'instruction package
```

```
public class JeuDeDes {  
    ...  
}
```

Fichiers jar

Les classes peuvent être compactées et stockées dans un fichier .jar

- Identique à une structure de fichiers .zip

Java peut récupérer des classes à partir d'un fichier .jar

- Les chemins internes doivent suivre les règles normales
- Spécifier le fichier .jar dans CLASSPATH (comme un répertoire racine)
 - CLASSPATH=c:\jod\mystuff.jar

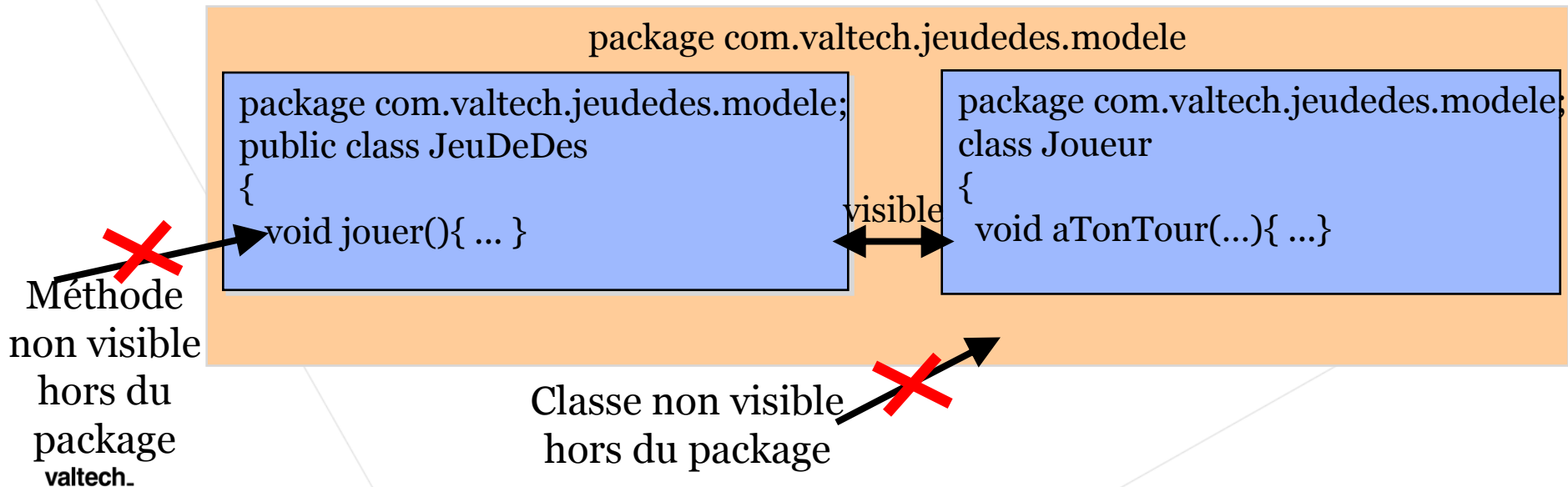
Les classes de l'API de base de Java résident dans :
`\jdk\jre\lib\rt.jar`

- Inutile de placer ceci dans CLASSPATH
- Automatiquement placé

Visibilité de niveau package — visibilité « par défaut »

Pas de mot-clé *public* ou *private*? La visibilité par défaut est « package »

- visible à toutes les classes du même package (s'applique aux méthodes, aux attributs, et aux classes elles-mêmes)



Création de noms de package

Par convention, les noms de package commencent par des minuscules pour se différencier des noms de classe

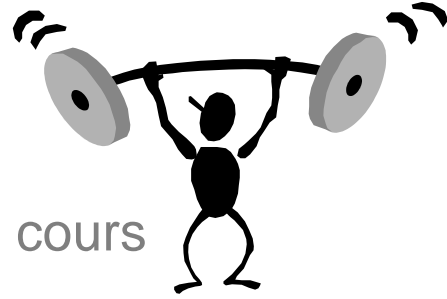
Pour éviter les conflits de noms, Sun conseille de nommer les packages à l'aide de votre nom de domaine Internet en en inversant les composants

- com.mutuelleagricole.finance
- com.banquecommerciale.finance
- com.unionbanquesSuisse.finance



La convention permet d'assurer l'unicité mondiale des noms de classe.

Exercice



Placez dans un package les classes de votre projet en cours

- Le formateur vous indiquera quel package utiliser

Java et la Conception Objet

Notions de processus de développement

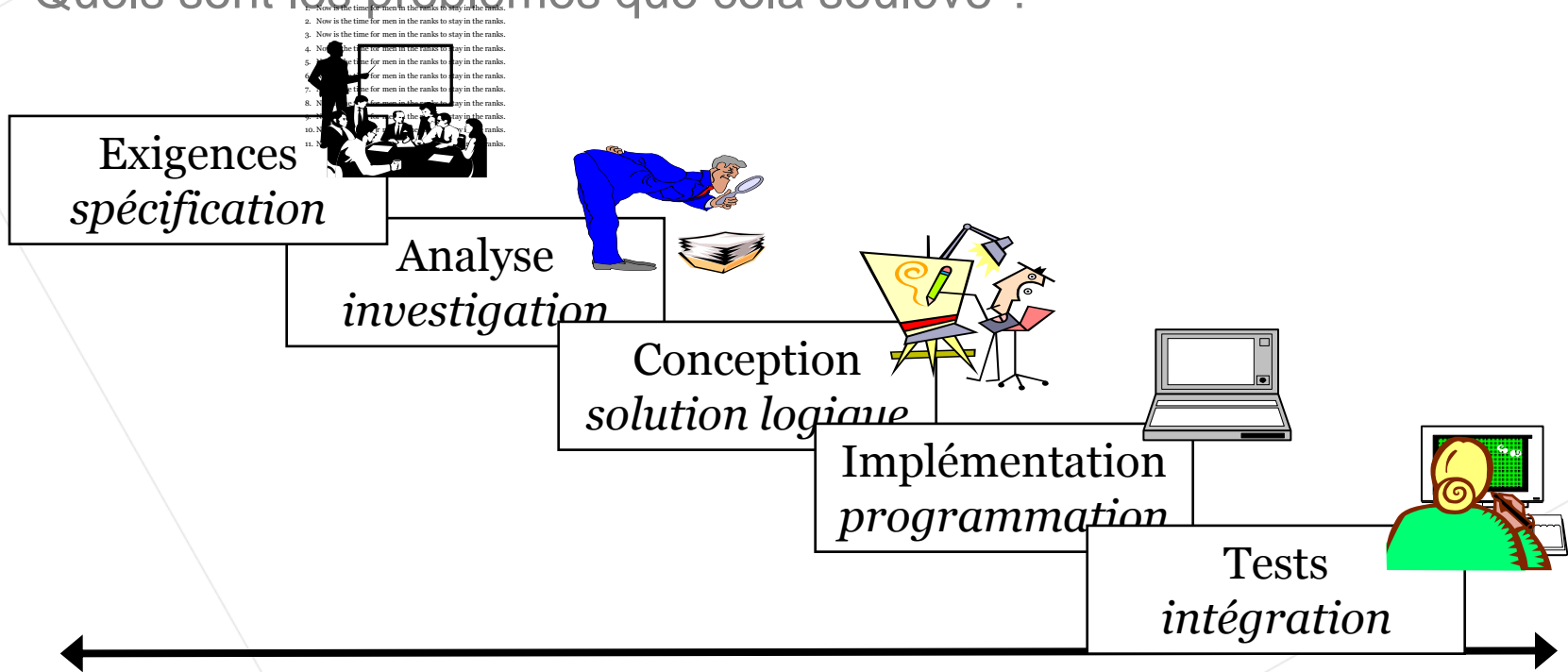
Objectifs

Comprendre comment formulation des exigences, analyse, conception, implémentation et tests s'intègrent dans un processus de développement logiciel itératif

Identifier le rôle d'UML (Unified Modeling Language)

Le cycle de développement en cascade

Quels sont les problèmes que cela soulève ?



Inconvénients de la cascade

Suppose que nous pouvons définir tous les besoins dès le début du projet

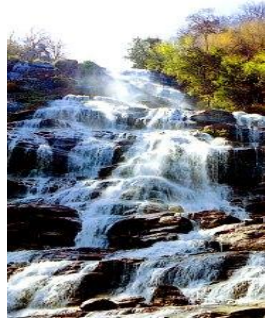
“Combat” le changement par un effort d’avoir
“bon dès le départ”, plutôt que de considérer la gestion des
changements comme une activité
de base du processus

Retarde la résolution des risques (ex. : intégration)

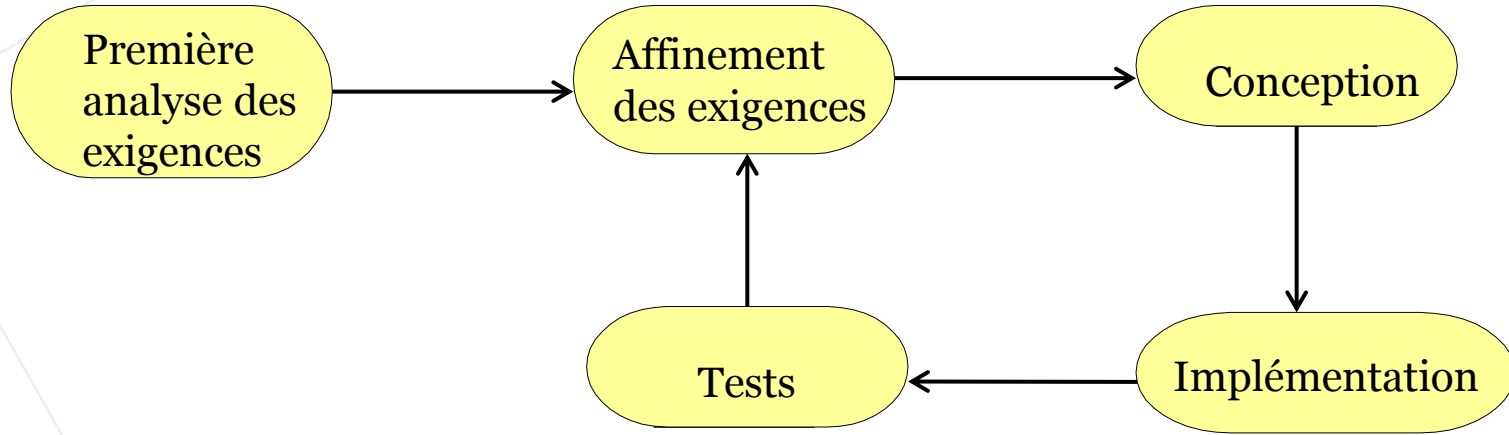
Est focalisé sur les documents et les revues

Amène à des conflits entre les différentes parties :

- manque de clarté dans la définition des exigences
- engagements importants en dépit de grandes incertitudes
- désir inévitable de modifications



Le cycle de vie itératif



On exécute ce cycle (appelé « itération ») plusieurs fois

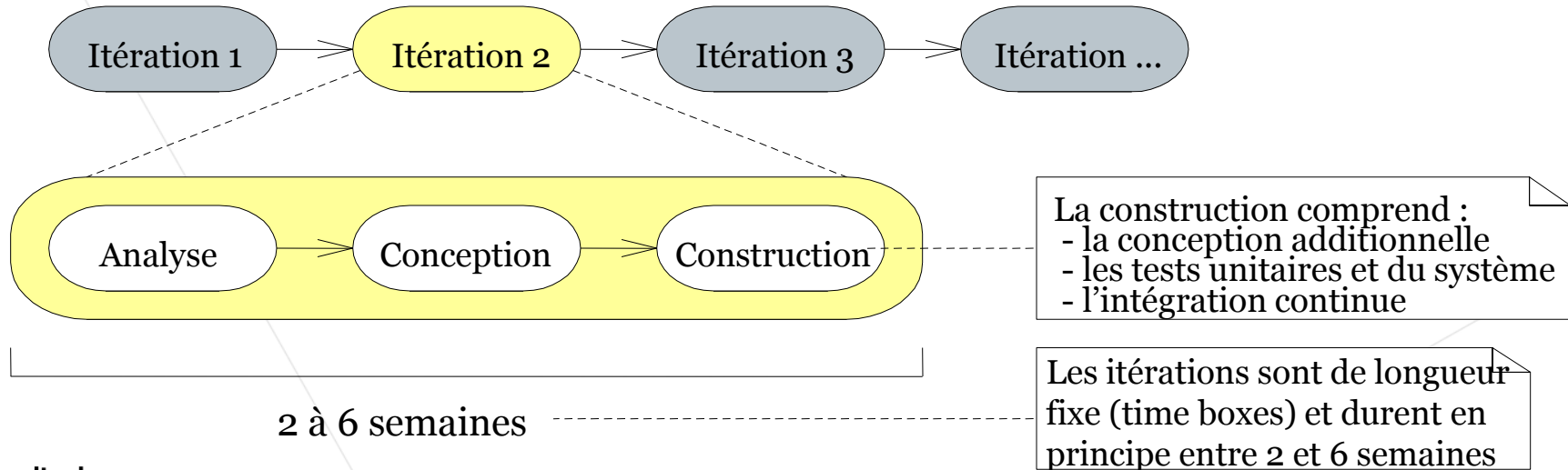
Chaque itération enrichit de façon incrémentale le système

Les technologies objet, extensibles et évolutives, s'y prêtent particulièrement bien

Développement logiciel itératif

Procéder par petites étapes (itérations), intégrer les réactions, affiner

À chaque itération, incrémenter le périmètre du système jusqu'à couvrir la totalité des exigences



UML (Unified Modeling Language)

Notation standard de l'industrie pour les modèles d'analyse et de conception orientées objet

UML propose de nombreux diagrammes, et nous avons déjà rencontré trois d'entre eux

- Diagramme de classes
- Diagramme de séquence
- Diagramme de communication

C'est un langage

- ni une méthode, ni un processus de développement

Résumé

Le développement logiciel itératif est un processus efficace, incrémental, par « petites étapes »

Les méthodes orientées objet conduisent à la création de systèmes extensibles, essentiels au développement itératif

UML (Unified Modeling Language) est une notation qui complète ces processus

Java et la Conception Objet

**Modèles du domaine (ou
d'analyse)**

Objectifs

Lire et créer des modèles du domaine

Appliquer les principes adéquats pour l'identification des classes, des attributs et des associations

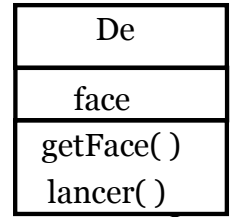
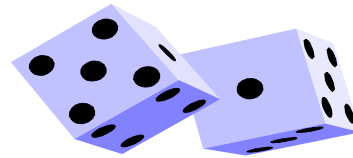
Ajouter aux modèles les attributs et les associations qui conviennent

Modélisation des concepts du domaine

Les systèmes orientés objet s'appuient sur des classes représentant les concepts du monde réel

Avant de faire des choix de solutions (conception logicielle), il faut bâtir un modèle du problème (analyse)

- Pour mieux comprendre le problème à résoudre
- Pour s'assurer de la prise en compte des règles métier
- Pour obtenir un modèle aussi indépendant que possible des choix technologiques

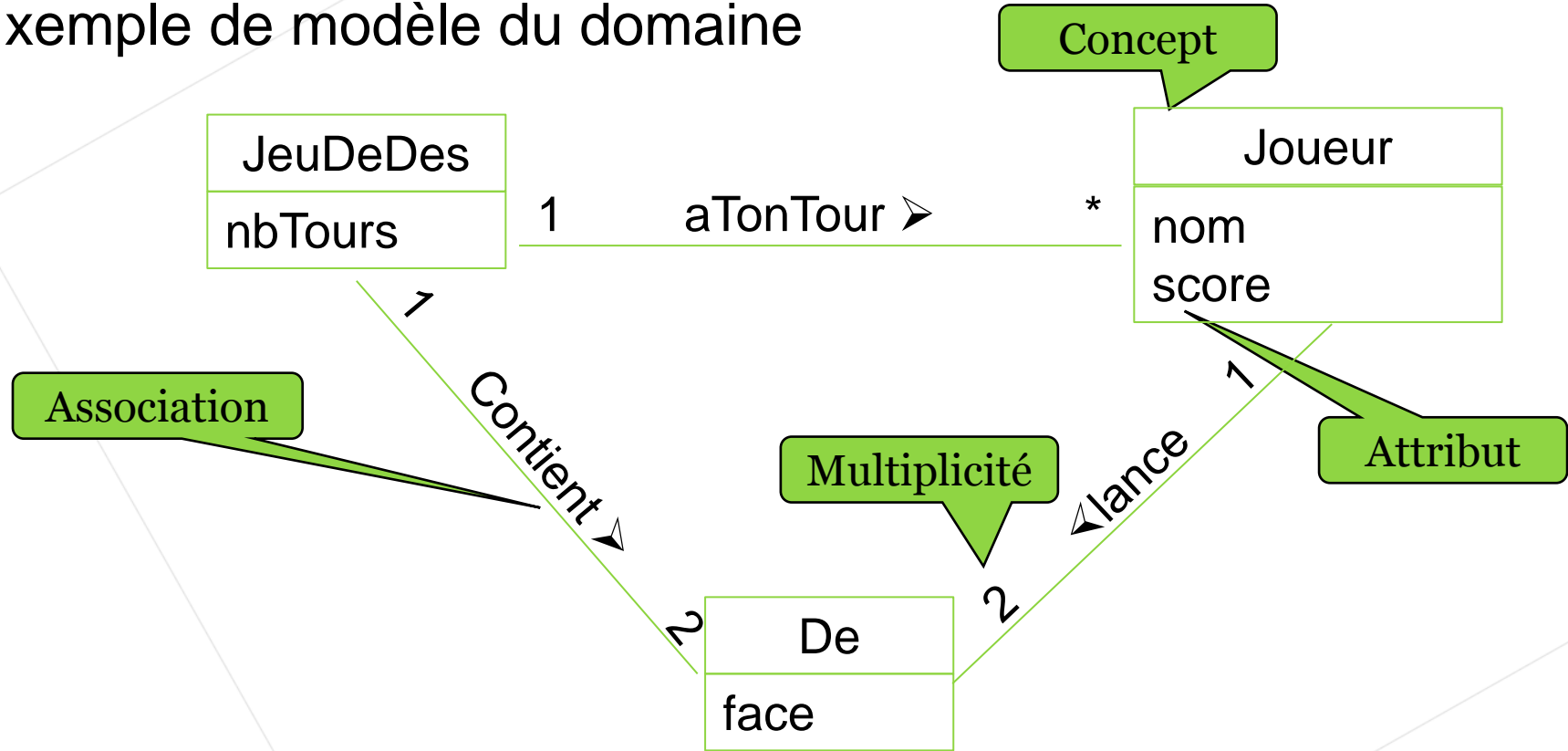


public class De
{
 private int face;

 public int getFace() {...}
 public void lancer(){...}
 //...
}

la création de classes
logicielles est une
activité de conception

Exemple de modèle du domaine



Les concepts dans les modèles du domaine

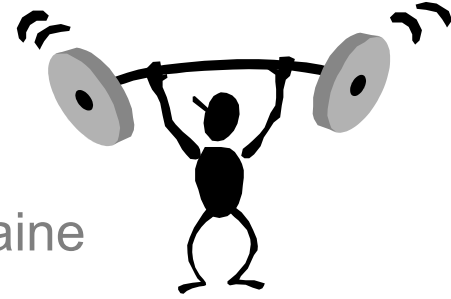
Les concepts sont des « éléments » du domaine du monde réel (concepts « métier »)

- Il peut s'agir de choses tangibles, physiques (dés, joueurs, ...)
- ou de choses abstraites, logiques

Exemple pour le Jeu de dés:

- Dés
- Joueurs
- Boîte du jeu, ...

Identification des concepts d'un domaine



La première étape de la création d'un modèle du domaine consiste à en identifier les concepts

Exercice :

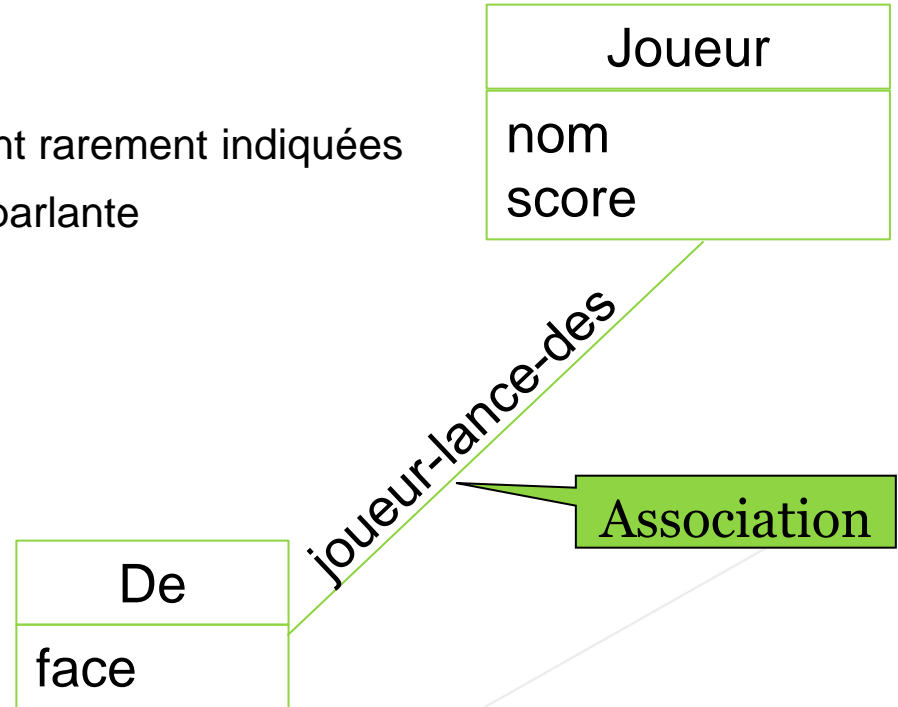
- créez une liste de concepts potentiels pour cycle 1 du développement du jeu de Monopoly
- Classez les dans les catégories du tableau suivant :

Catégorie de concepts	valeurs
Objets tangibles	
Rôles des individus	
Conteneurs	
Articles des conteneurs	

Les associations

Les associations constituent des relations dignes d'intérêt reliant des concepts

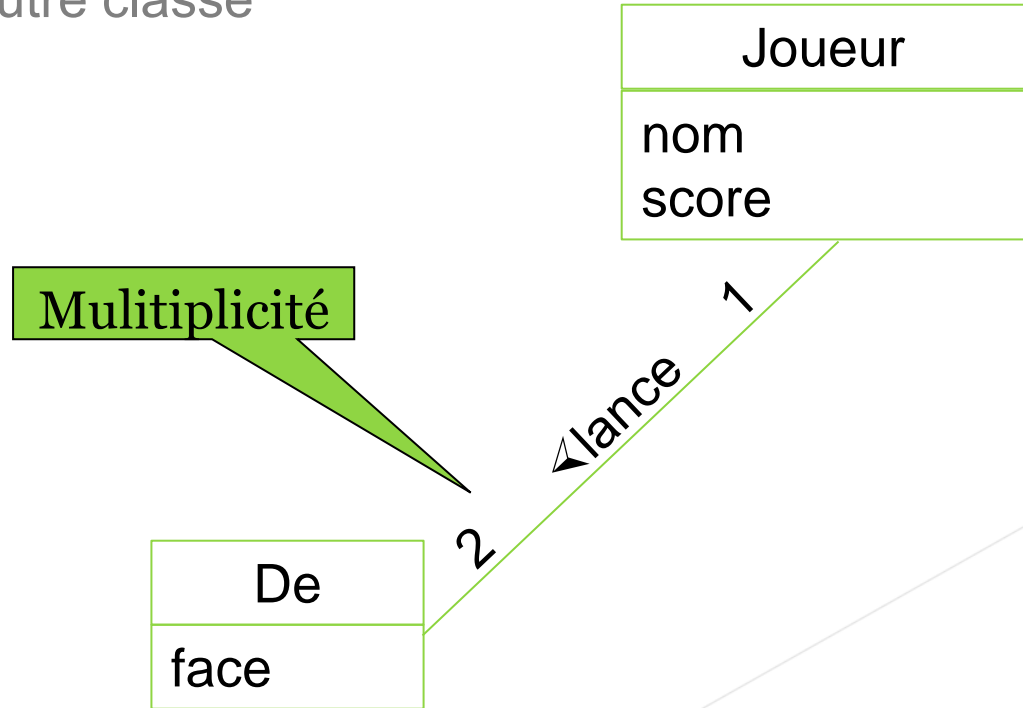
- Elles sont bidirectionnelles
- Les relations brèves, transitoires sont rarement indiquées
- Utilisez une forme nom-verbe-nom parlante
 - Adhérent Emprunte Ressources
 - Catalogue Recense Livres



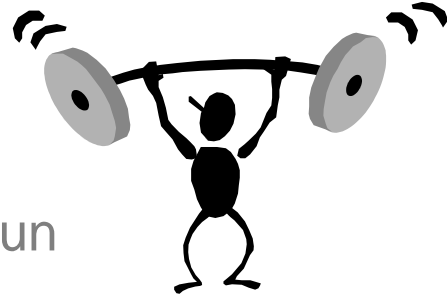
La multiplicité

La multiplicité représente le nombre d'objets d'une classe donnée liés à un objet d'une autre classe

- * (zéro ou plus)
- 0..* (zéro ou plus)
- 0..2 (zéro à deux)
- 1..* (un à plusieurs)
- 40 (nombre exact)



Exercice



Utilisez les listes de concepts potentiels pour élaborer un modèle du domaine

Faites apparaître les concepts, les associations et la multiplicité

Les attributs

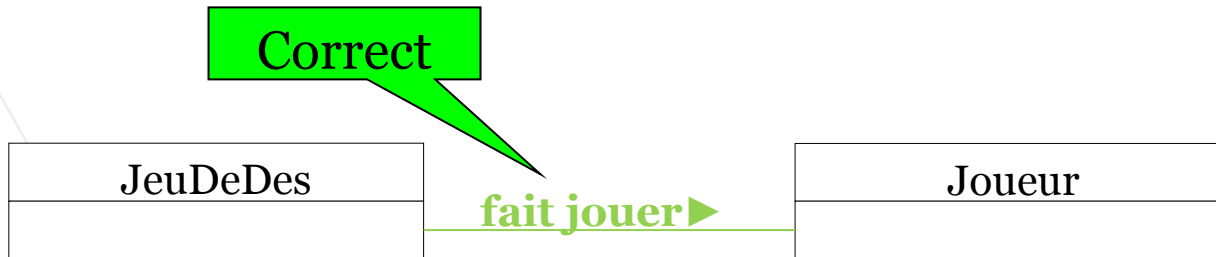
Les attributs sont des données primitives qui appartiennent à des concepts du domaine ou permettent de les décrire

- Ils doivent être représentés par des types simples
 - ex. : nombre, string, booléen, date, heure...
- Permettent de répondre aux besoins d'informations des exigences

De
face

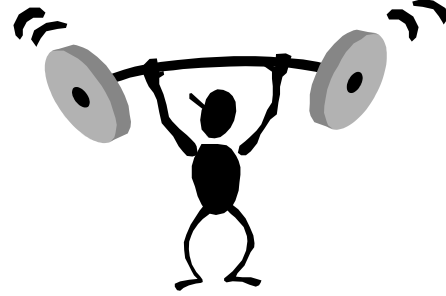
Attention aux « clés étrangères » !

Liez les concepts par une association, et non par un attribut



Exercice

Ajoutez les attributs au modèle du domaine



Résumé

Les modèles du domaine illustrent

- les concepts du monde réel (objets métier)
- leurs attributs
- leurs associations logiques

Java et la Conception Objet

Vers le modèle de conception

Objectif

Comprendre la transition Analyse / Conception

- De la modélisation du problème vers la solution

Apprendre les principes clés de la conception et les enjeux associés

- But: concevoir un système simple à maintenir, flexible et dont les composants peuvent être réutilisés

Présentation

Le modèle de conception est déduit du modèle d'analyse (de domaine)

- Les classes métier existantes sont détaillées, parfois divisées
- De nouvelles classes sont créées à caractère purement informatique (IHM, persistance des données,...)

L'élaboration du modèle de conception compte deux tâches clés:

- Attribuer des responsabilités aux objets
- Concevoir des interactions entre les objets

C'est un enjeu prépondérant

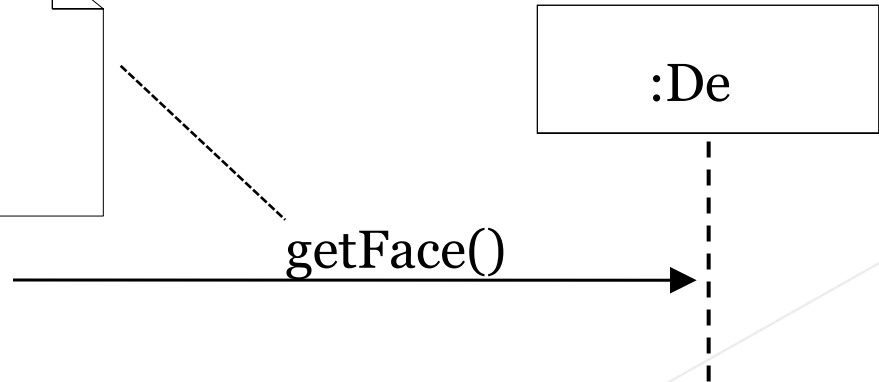
- Exerce une influence majeure sur la qualité du logiciel (maintenabilité, flexibilité, réutilisabilité)

L'affectation de responsabilités

Responsabilité : savoir ou savoir faire

- liée aux méthodes et aux données des classes
- remplie par une ou plusieurs méthodes
- éventuellement répartie sur plusieurs classes

L'envoi du message
getFace implique que
dé a la responsabilité de
connaître sa face



Principes de base

Il existe des principes universels quant à l'affectation des responsabilités

- Ils ont été réunis sous le nom de patterns GRASP (General Responsibility Assignment Software Patterns)

Dans ce chapitre, nous allons voir les principes prépondérants

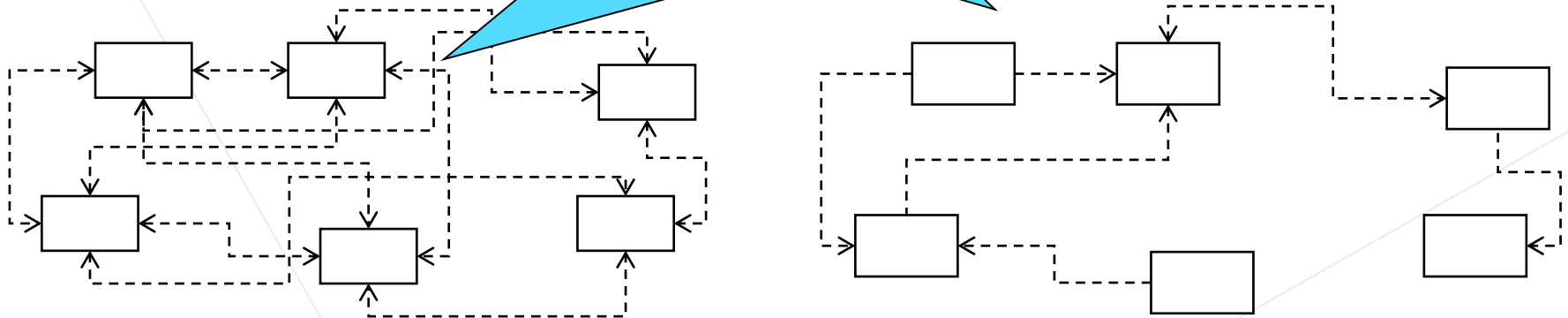
- Faible couplage
- Forte cohésion
- Expert en information
- Créateur

Faible couplage (1/2)

Couplage

- Mesure le degré de dépendance d'un module (classe, ou ensemble de classes) vis-à-vis d'autres modules

Lequel de ces systèmes résiste le mieux au changement ?
Est fait de composants réutilisables?



Faible couplage (2/2)

Le faible couplage est un pattern évaluatif, qu'un concepteur applique de façon continue

- Certains outils mesurent automatiquement le niveau de couplage d'un système

Particulièrement important lorsque le composant dont dépend un autre est ou risque d'être instable !

- Par exemple, le couplage fort avec les types du package `java.lang` est acceptable ; ils sont susceptibles d'être stables, et présents pour de nombreuses années

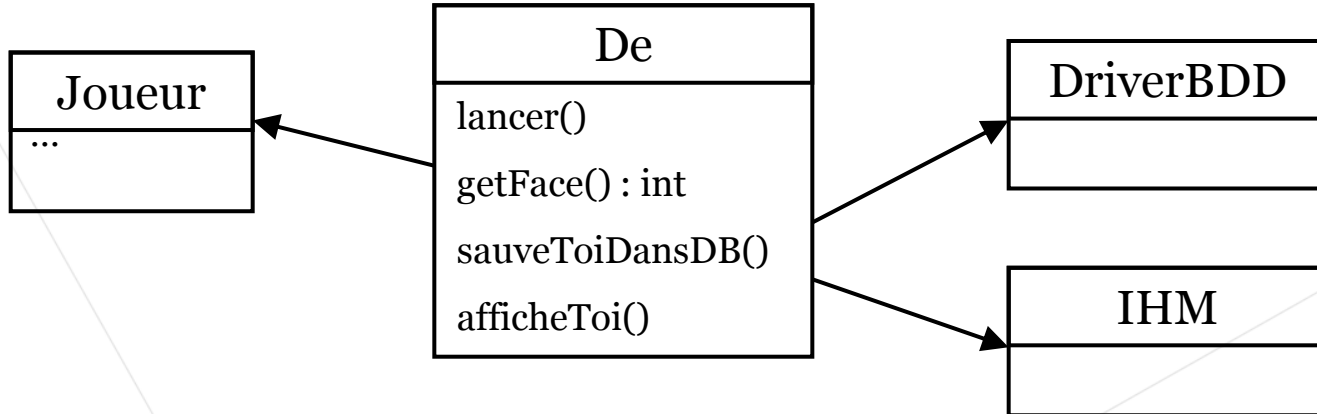
Forte cohésion (1/2)

Cohésion

- Mesure du degré de cohérence des responsabilités (méthodes et données) d'une classe

Ce principe est intimement lié au précédent

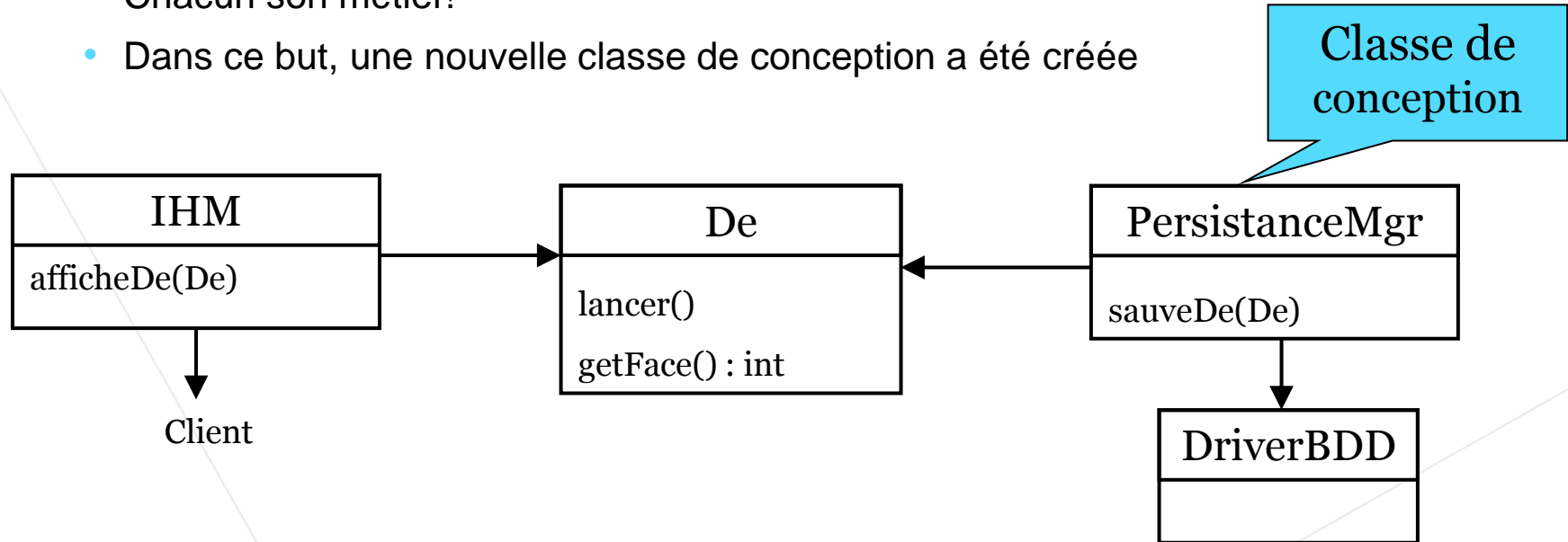
- Une classe ayant des responsabilités disparates sera liée à de nombreuses autres classes, de toutes sortes



Forte cohésion (2/2)

Plus de cohésion / moins de couplage

- Chacun son métier!
- Dans ce but, une nouvelle classe de conception a été créée



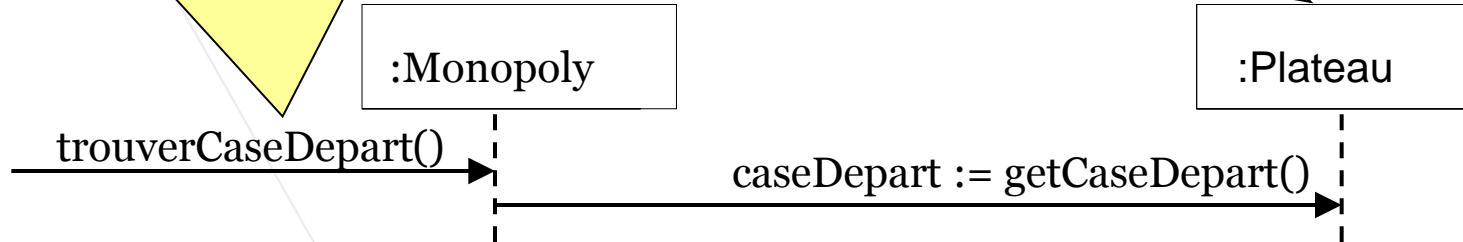
Expert en informations

Quel est le principe d'affectation des responsabilités le plus généraliste ?

- Assignez une responsabilité à l'expert en informations – la classe ayant les informations nécessaires pour remplir cette responsabilité

Nous avons besoin de trouver la case départ. Qui doit avoir la responsabilité de fournir cette information ?

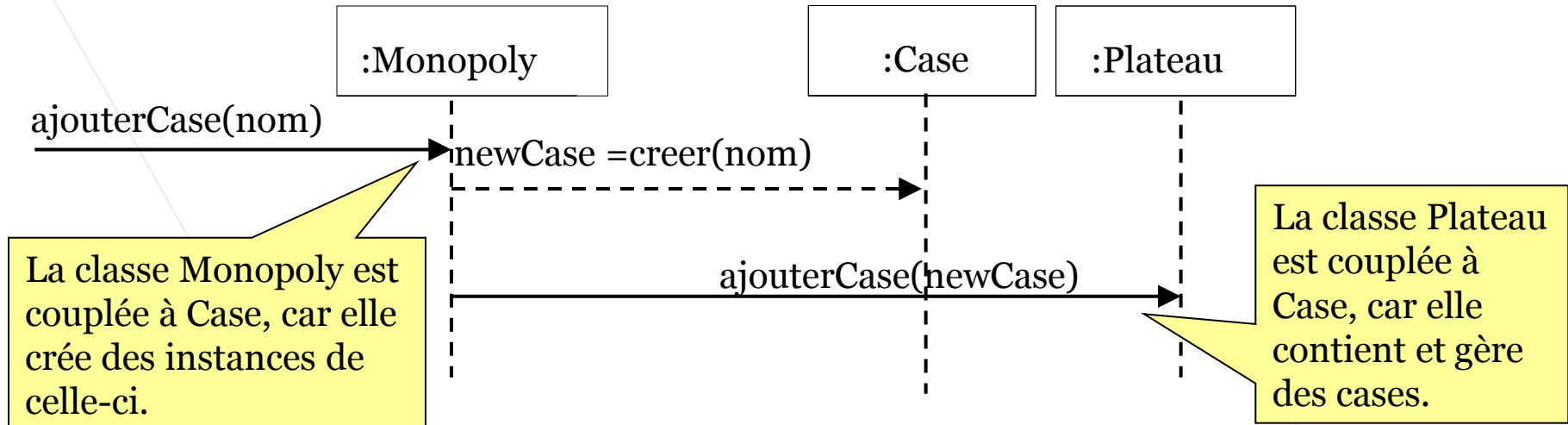
Le plateau est l'expert en informations sur les cases. Il les contient.



Créateur (1/2)

La création d'une instance d'une classe exige d'avoir quelque connaissance de cette classe – couplage

- Si la classe A crée une instance de la classe Z, A est couplée à Z

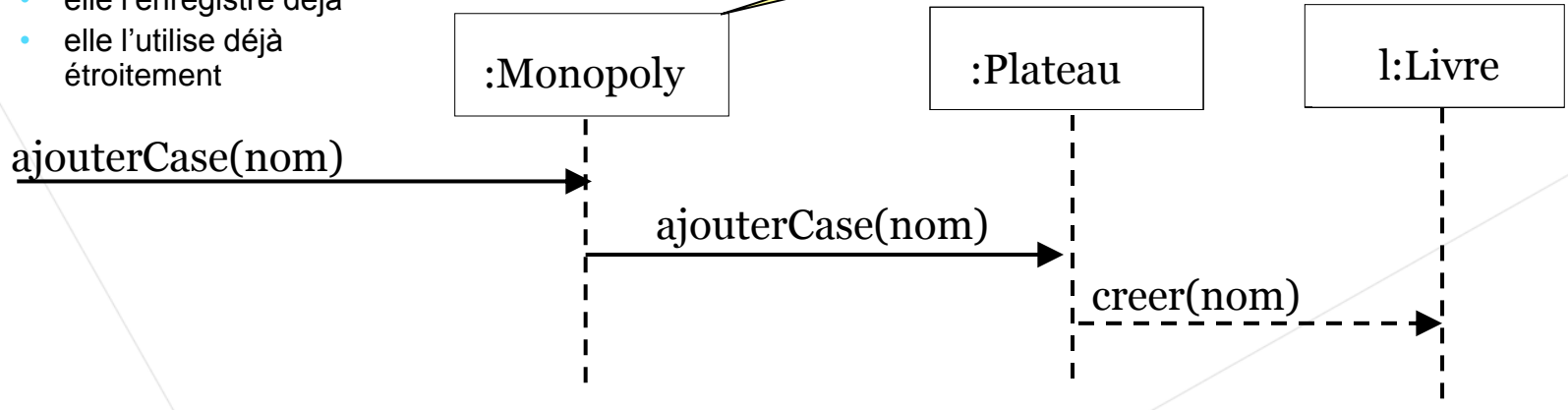


Créateur (2/2)

Comment assigner la responsabilité de créer une instance sans accroître inutilement le couplage ?

- Appliquez le pattern Créateur
- La classe A doit créer une instance de la classe Z si :
 - elle la contient déjà
 - elle l'agrège déjà
 - elle l'enregistre déjà
 - elle l'utilise déjà étroitement

La classe Monopoly n'est pas couplée à Case



Résumé

L'affectation des responsabilités est un point clé de la conception orientée objet

De nouvelles classes « pratiques » apparaissent en conception (non issues du domaine métier)

On s'efforce d'écrire des classes les plus indépendantes possibles, et constituées de responsabilités cohérentes

Par ce moyen, on obtient des systèmes faciles à maintenir, à faire évoluer, et réutilisables

Java et la Conception Objet

**Modèle dynamique -
diagrammes d'interactions**

Diagrammes d'interactions UML

Les diagrammes d'interactions

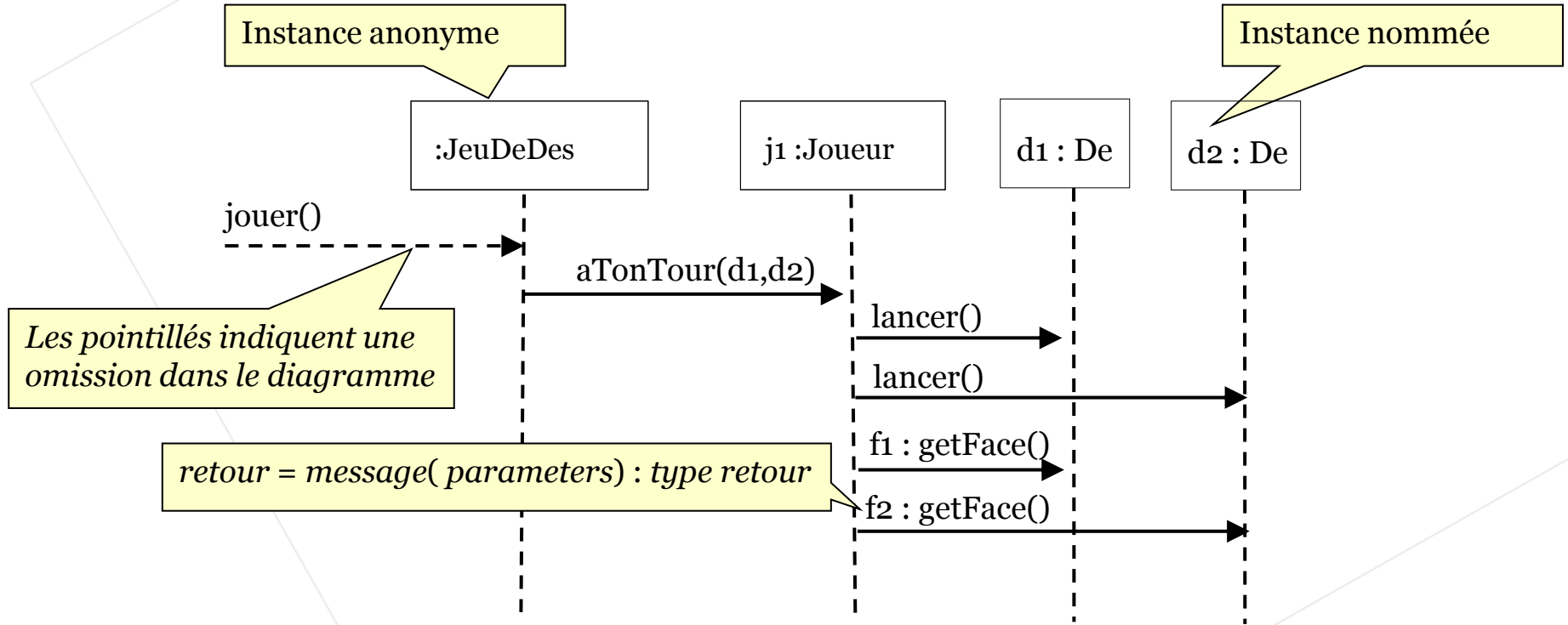
- Rendent compte des messages et des interactions entre objets

UML en fournit deux types

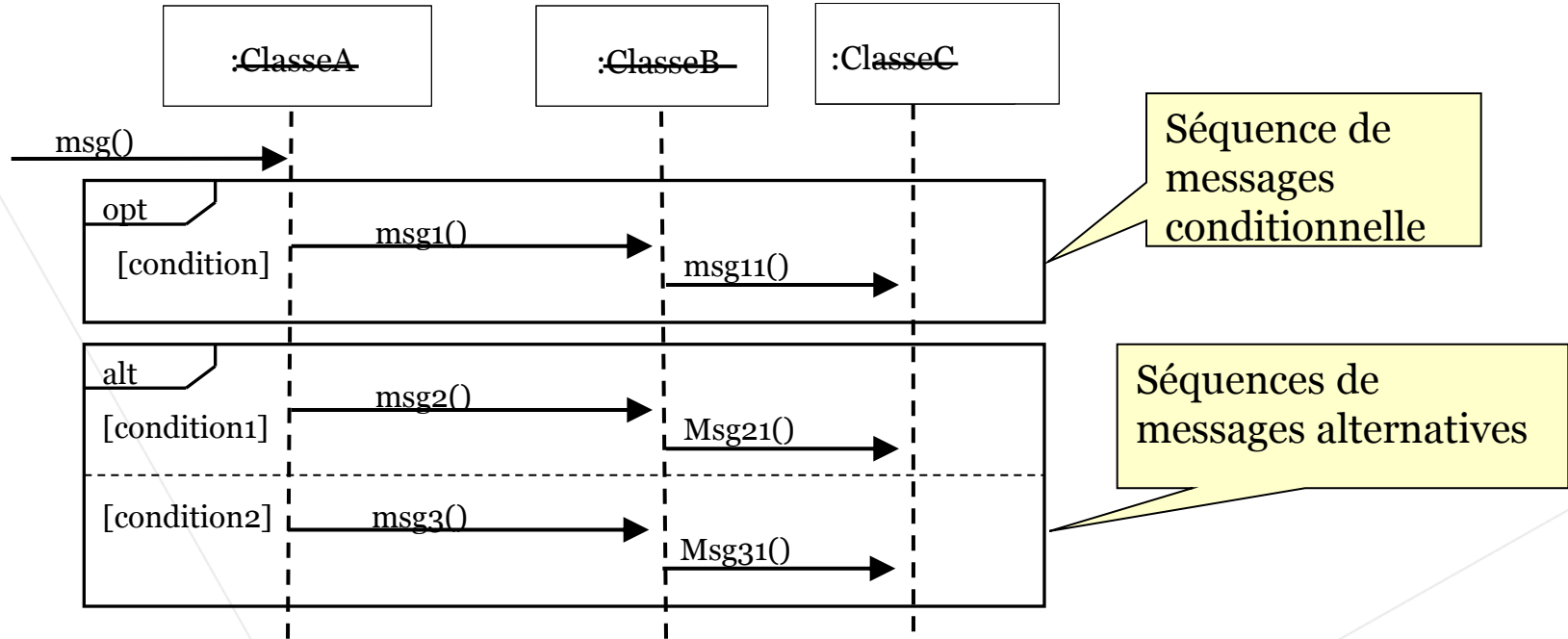
- Le diagramme de communication reflétant à la fois les échanges de messages et les relations entre objets
- Le diagramme de séquence qui proposent un déroulement chronologique facile à suivre

Nous privilégierons les diagrammes de séquence, plus intuitifs

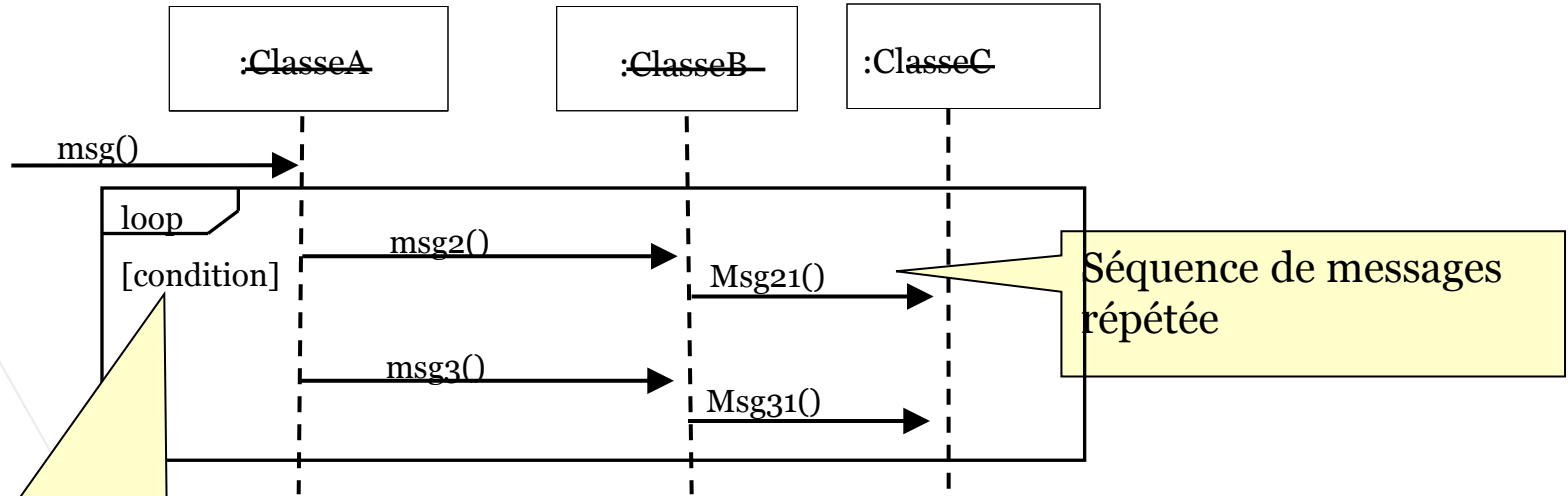
Les diagrammes de séquence



Séquences conditionnelles

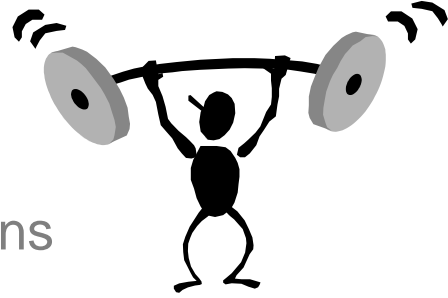


Séquence répétées (boucles)



Ex: [pour chaque joueur] ou
[1..N] (de 1 à N)

Exercice



Créez des diagrammes de séquence pour les opérations système que vous indiquera le formateur

- suggestion : « créer le jeu » et « jouer »
- Pensez à appliquer les principes de conception (faible couplage / forte cohésion, ...) et discutez de vos choix

Résumé

Les diagrammes d'interactions décrivent les messages qu'échangent les objets pour remplir leurs tâches

Les diagrammes de communication et de séquence constituent deux types de diagrammes d'interactions UML

- Les diagrammes de communication représentent à la fois les liens entre les objets et les messages qu'ils échangent
- Les diagrammes de séquence mettent en évidence l'enchaînement chronologique des messages

Java et la Conception Objet

Diagrammes de classes de conception

Objectifs

Lire et créer des diagrammes de classes de conception

Ajouter des méthodes au diagramme

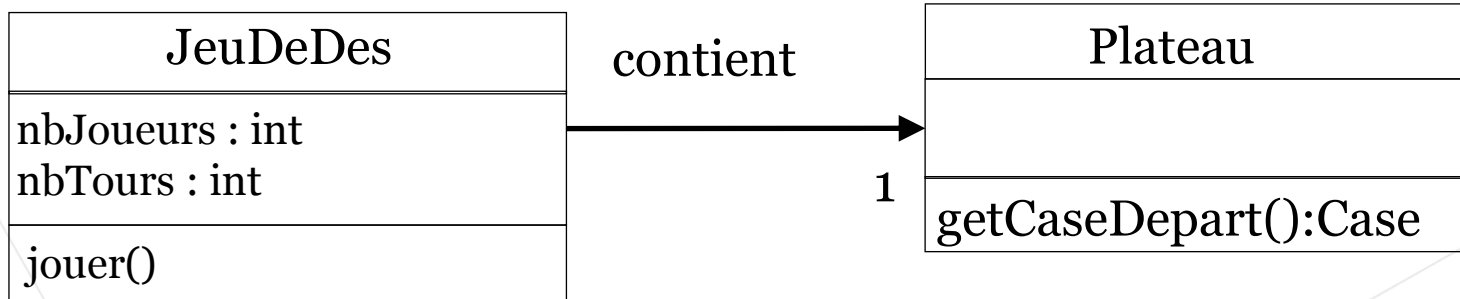
Rendre compte de la visibilité

Définir et faire ressortir les attributs de référence et les attributs simples

Diagrammes de classes de conception

Les diagrammes de classes de conception décrivent la structure des classes logicielles

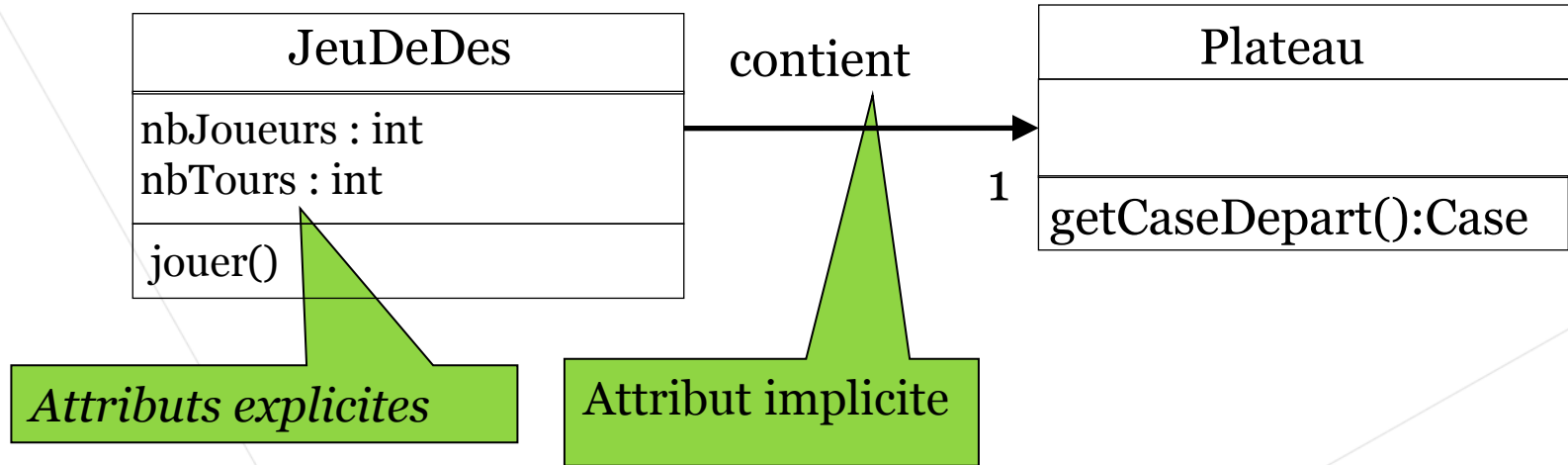
- Noms de classe
- Attributs (explicites et implicites)
- Méthodes



Les attributs explicites et implicites

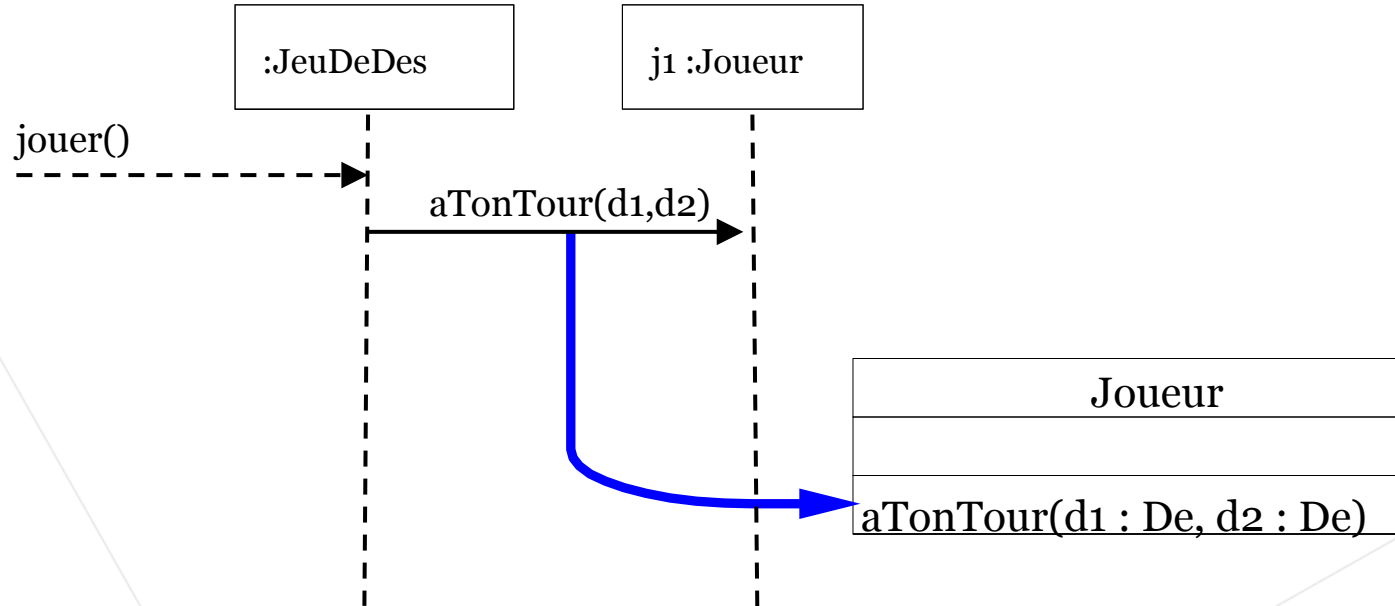
Montrez les attributs simples, explicites dans le compartiment de la classe

Montrez les attributs de référence, implicites à l'aide d'une flèche de navigation UML



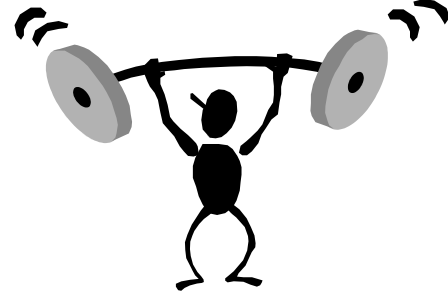
Les méthodes

Identifiez les méthodes sur les diagrammes de séquence



Exercice

Concevez un diagramme de classes de conception pour le projet du cours



Java et la Conception Objet

**Mapping des artefacts de
conception avec le code**

Objectifs

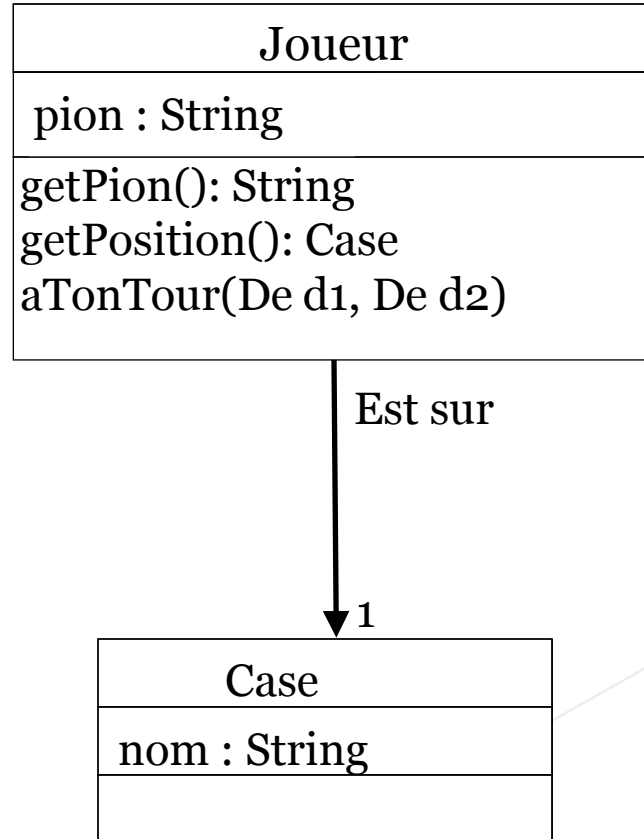
Traduire des diagrammes de classes de conception en définitions de classes

Traduire des diagrammes de communication en méthodes

Choisir l'ordre d'implémentation des classes

Définition de classes (1/2)

Quel code écririez-vous pour la classe Joueur ?



Définition de classes (2/2)



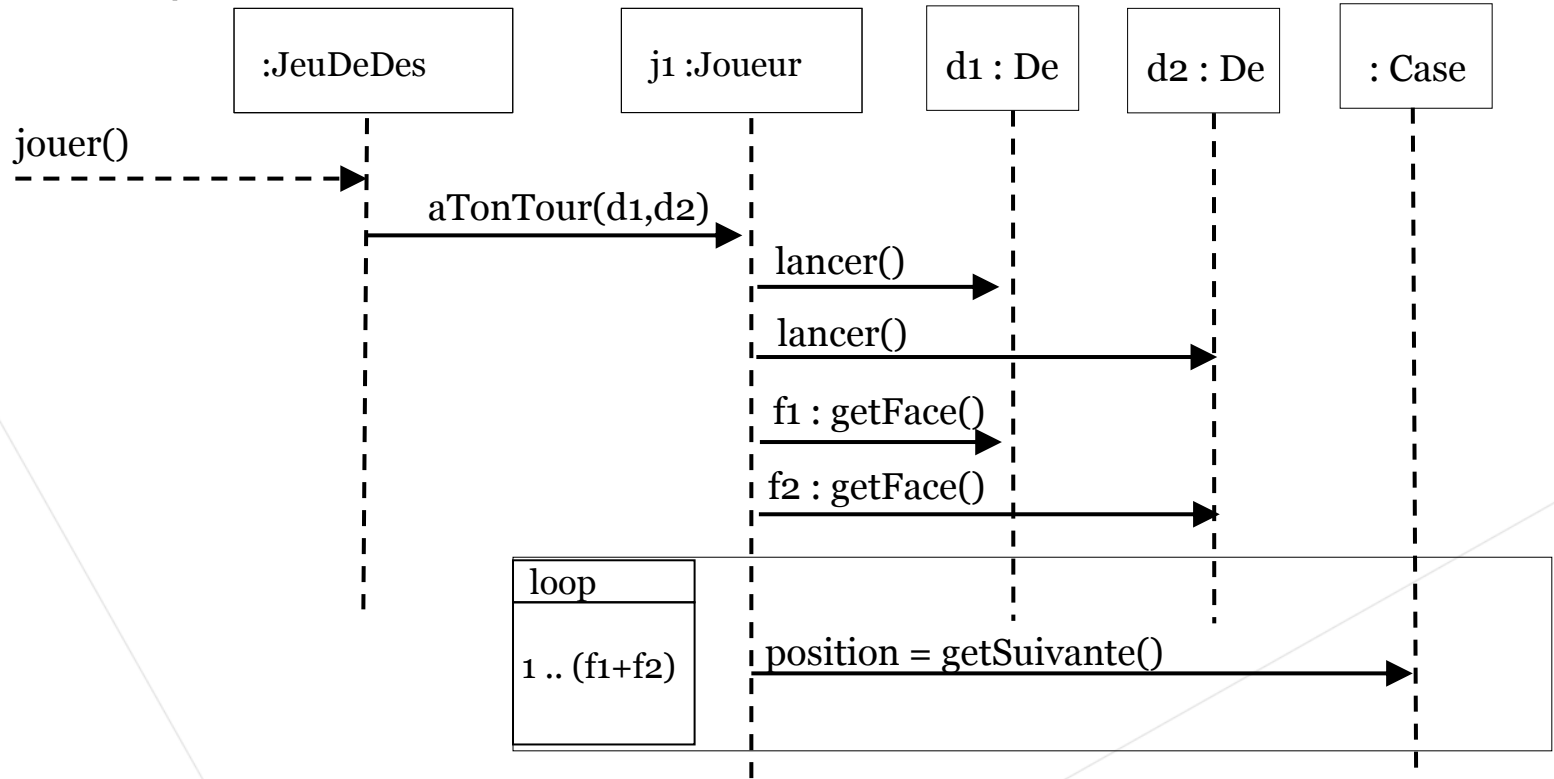
```
package com.valtech.jeudedes.modele;

public class Joueur{
    private String pion;    //attribut direct
    private Case position; //attribut par agrégation

    public String getPion() {
        return pion;
    }
    public Case getPosition() {
        return position;
    }
    public void aTonTour (De d1, De d2) {
        ...
    }
}
```

Définition de méthodes (1/2)

Quel code pour aTonTour ?



Définition de méthodes (2/2)



```
package com.valtech.jeudedes.modele;

public class Joueur{
    ...
    public void aTonTour (De d1, De d2) {
        d1.lancer();
        d2.lancer();
        int f1 = d1.getFace();
        int f2 = d2.getFace();
        for(int i = 0; i < f1+f2 ; i++){
            position = position.getSuiivante();
        }
    }
}
```

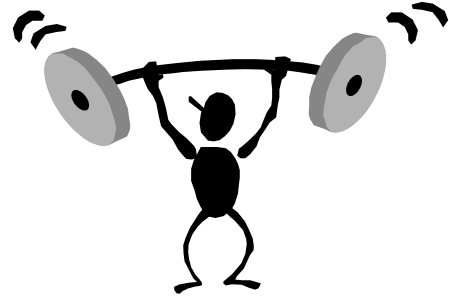

Stratégie d'implémentation

Commencez par la classe la moins connectée pour aller vers celle qui est la plus connectée

Implémentez et testez chaque classe de façon complète avant de passer à la suite

Exercice

Écrivez les définitions de classes et les méthodes selon les instructions du formateur



Résumé

Mapping avec le code

- Relativement direct à partir des artefacts de conception
- La réalité ne sera pas aussi « nette »
- Il reste beaucoup de marge pour les décisions de conception mineures lors de la programmation

Réflexion

- Degré de créativité pendant la phase de codage ?
- Outils AGL pour la génération de code ?

Java et la Conception Objet

Les concepts fondamentaux

Objectifs

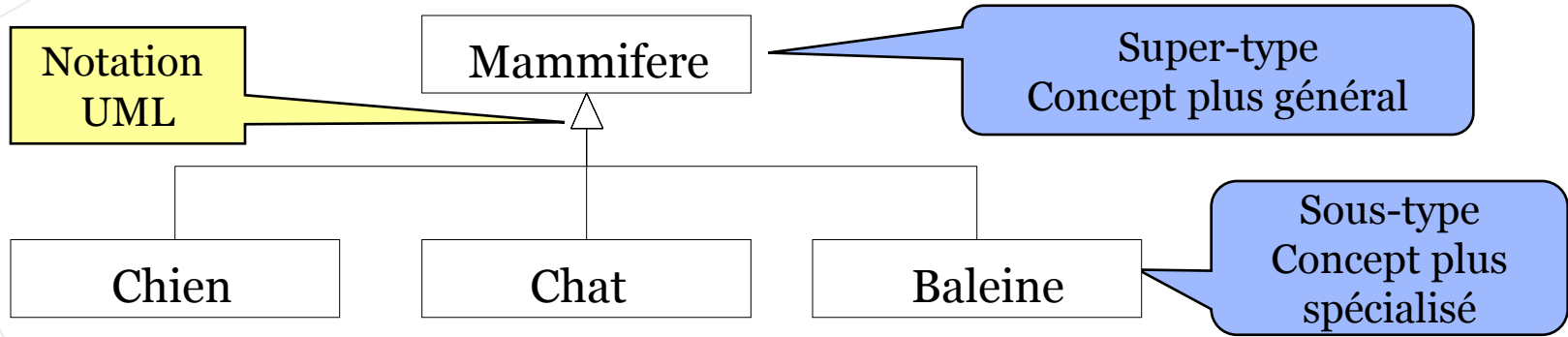
Comprendre les notions de généralisation-spécialisation et d'héritage

Comprendre le polymorphisme

Coder des hiérarchies d'héritage (super-classes et sous-classes)

Coder et utiliser des méthodes polymorphes

Hiérarchie de généralisation-spécialisation

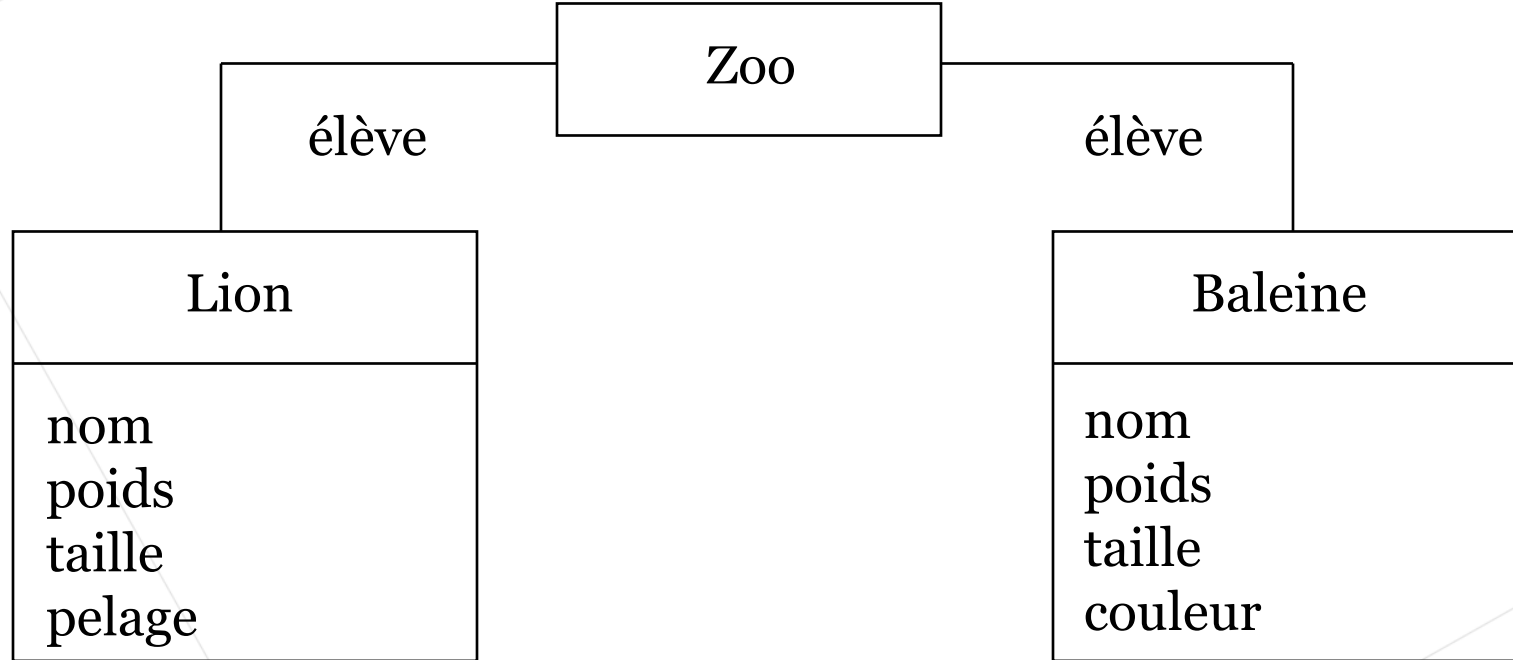


Une classe possède des caractéristiques

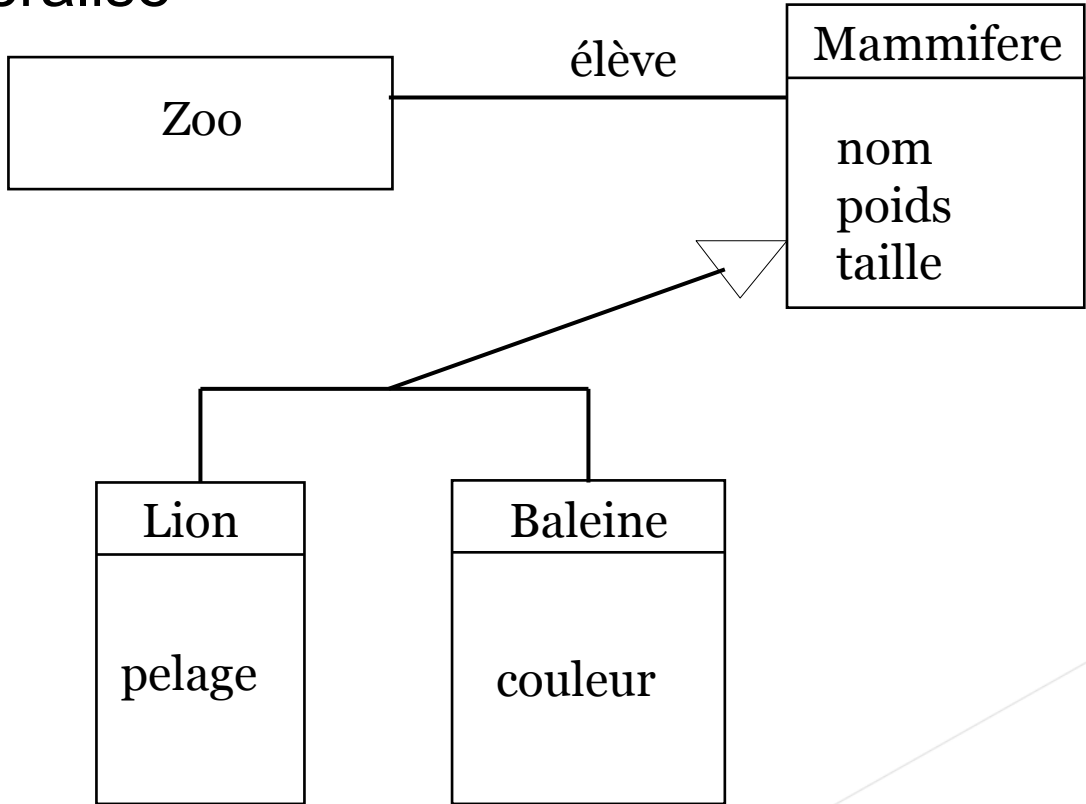
- attributs
- associations
- comportement (méthodes)

Les caractéristiques communes sont placées dans la super-classe et les spécifiques dans la sous-classe

Exemple



Ajout d'un concept généralisé



Règles

Une spécialisation potentielle S d'une généralisation G doit réussir ces deux tests simples :

Test « est-un »

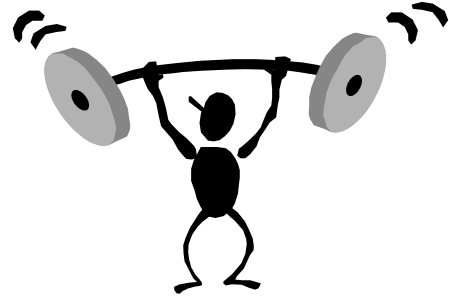
- Cela a-t-il un sens de dire que « S est une sorte de G » ?
- Conformité au type d'adhésion défini

Test « 100% »

- Cela a-t-il un sens, pour S, de se conformer à 100% à la définition de G en termes d'attributs, d'associations et de traitement ?
- Conformité à la définition

Exercice

Créez le modèle d'analyse (de domaine) du cycle 2 du développement du Monopoly



Extension de la classe De

Les objets peuvent présenter des variations sur un même thème

Un dé pipé favorise un côté

- La face prend la valeur de la face pipée dans la moitié des lancements du dé

DePipe
face facePipee
lancer() getFace() setFacePipee()

De et DePipe : parties en commun

DePipe et De partagent des éléments communs :

- Ils ont tous deux un attribut face
- `getFace()` utilise le même code
- `lancer()` est commun aux deux, mais fonctionne différemment

De
face
lancer() getFace()

DePipe
face facePipee
lancer() getFace() setFacePipee()

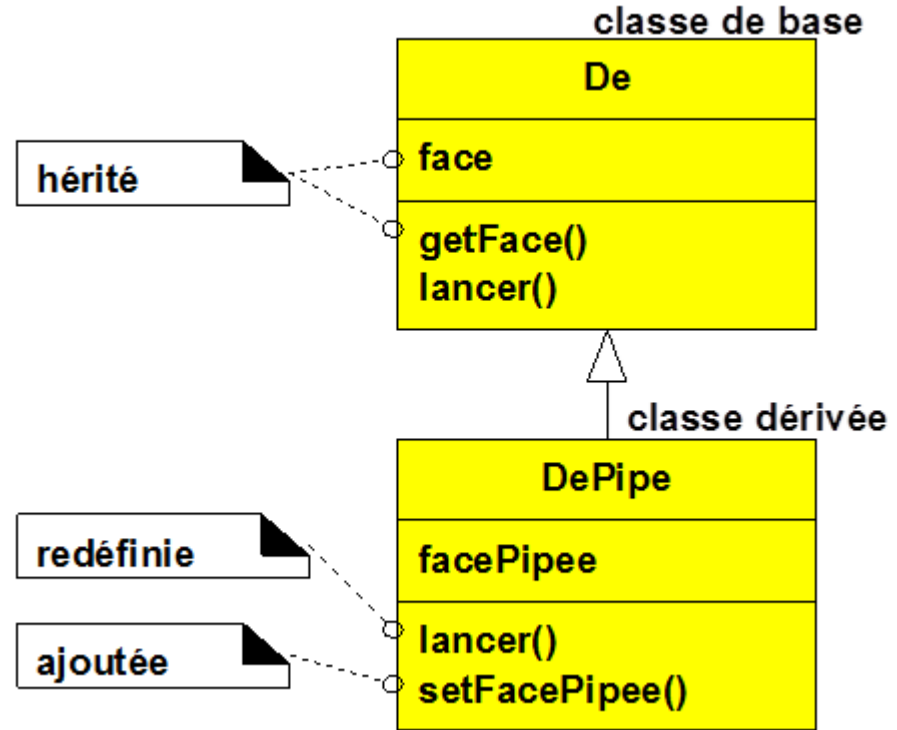
L'héritage

L'héritage est un mécanisme permettant la réutilisation du code

- Les sous-classes acquièrent tous les attributs et les définitions de méthodes de leur super-classe
- Les sous-classes peuvent redéfinir les méthodes héritées

La super-classe est également appelée classe « de base » ou classe « généralisée ».

La sous-classe est également appelée classe « dérivée » ou classe « spécialisée ».

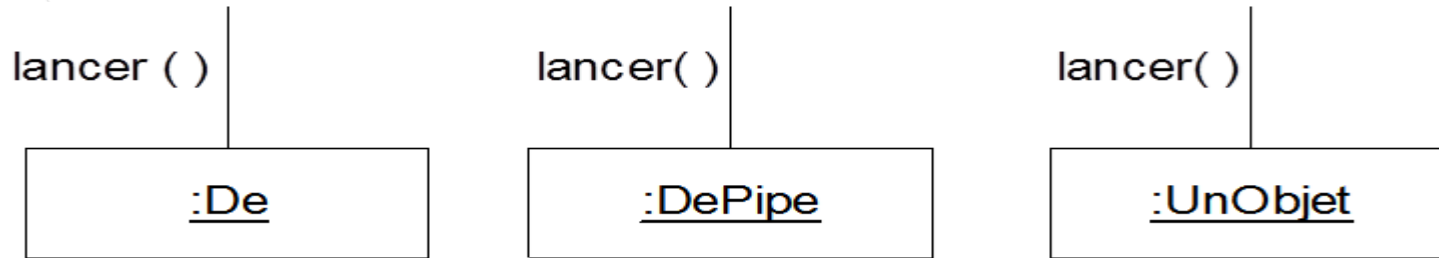


Polymorphisme

Le même message peut être envoyé à différents types d'objets, mais c'est l'objet qui détermine la réponse au message

Le programmeur n'a pas besoin de connaître la classe exacte de l'objet auquel est envoyé le message

- L'objet connaît le moyen approprié de répondre au message



Polymorphisme — Et alors ?

Utilisation de code non orienté objet ; plusieurs types de De à lancer

```
public void lancerTous( De[] des ) {  
    for(int i=0; i < des.length(); i++) {  
        De d = des[i];  
        if( d.type == DE )  
            d.face = (int)Math.random()*6 + 1;  
        else if( d.type == PIPEE ) {  
            d.face = (int)Math.random()*6 + 1;  
            if( Math.random( ) > 0.5 )  
                d.face = d.facePipee  
        } ...  
    }  
}
```

Test du type

Si l'on ajoute un nouveau type de classe De, quels seront les changements au code ?

Orientation objet et polymorphisme

Lancement des dés de façon polymorphe

```
public void lancerTous(De[] des ) {  
    for(int i=0; i < des.length; i++) {  
        des[i].lancer();  
    }  
}
```

Si l'on ajoute un nouveau type de classe De, quels seront les changements au code ?

- Rien à faire car le polymorphisme implique de ne jamais tester le type d'un objet.

Polymorphisme et attribution des responsabilités

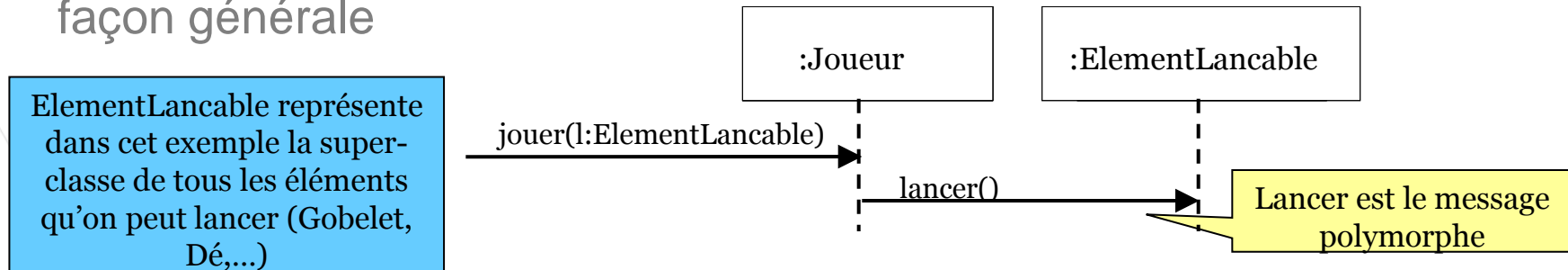
Les comportements varient en fonction du type?

Attribuez la responsabilité à la classe dans laquelle ce comportement diffère !

- Chaque classe pouvant faire rouler les objets d'une façon différente devra disposer d'une méthode lancer
- La classe elle-même sait comment elle doit agir
 - Renvoyer un nombre aléatoire de 1 à 6 (classe De)...
 - Ou bien favoriser une face plombée ? (classe DePipe)

Messages polymorphes

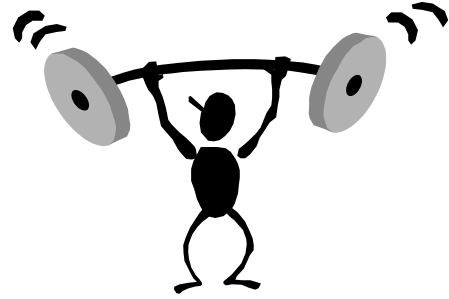
Dans les diagrammes de séquence, traitez les super-types de façon générale



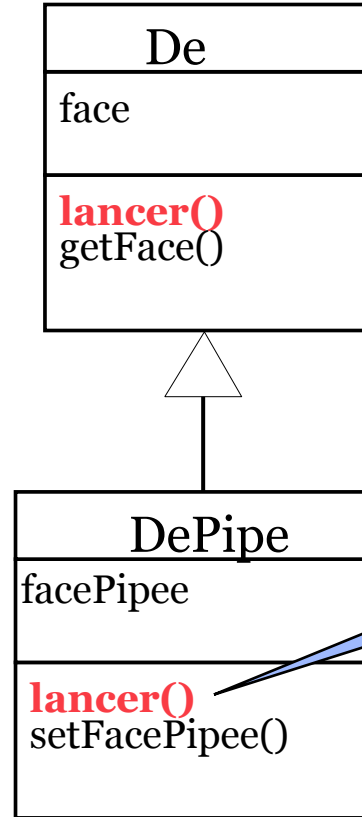
Déléguez les comportements spécifiques aux sous-types avec des zooms quand il y a une complexité dans le code métier.

Exercice

Créez les diagrammes de séquence du cycle 2 du développement du Monopoly



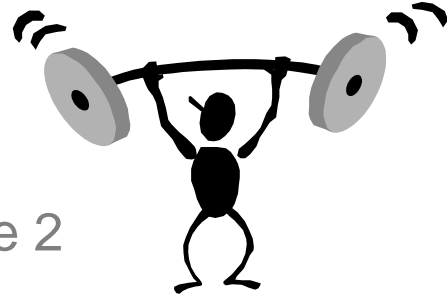
Diagrammes de classes et héritage



La méthode polymorphe doit apparaître dans la super-classe.

Exercice

Créez le diagramme de classes de conception du cycle 2 du développement du Monopoly



Création de sous-classes en Java

Sous-classe

Super-classe

```
class DePipe extends De {  
    public void setFacePipee( int fp ) {  
        facePipee = fp;  
    }  
  
    //clip...  
    private int facePipee;  
}
```

Ajout de nouvelles
méthodes

Ajout de nouveaux
attributs

Redéfinition de méthodes

Une méthode d'une sous-classe ayant la même signature qu'une méthode d'une super-classe remplace/redéfinit la méthode de la super-classe

Remplace la méthode lancer() de la super-classe.

```
public void lancer( ) {  
    super.lancer( );  
    if ( Math.random( ) > 0.5 )  
        setFace ( facePipee );  
}
```

Invoque la méthode lancer() de la super-classe.

Utilisez super pour invoquer la méthode de base

- Usage courant lorsque la méthode de la sous-classe étend ou s'ajoute à la méthode de la super-classe

Accès aux attributs des super-classes

Pourquoi utiliser la méthode `setFace()` dans `lancer()` au lieu d'accéder directement à `face` ?

```
public void lancer() {  
    super.lancer();  
    if ( Math.random() > 0.5 )  
        face = facePipee;  
}
```

Erreur. Pourquoi ?

Une sous-classe ne peut accéder aux attributs privés d'une super-classe

Pour cela, utiliser la visibilité « `protected` »

- Les méthodes des sous-classes ont accès aux membres protégés de la classe.

Membre protégé - exemple

Protégez `setFace()` oblige les classes extérieures à utiliser `lancer()` pour modifier la face

```
class De {  
    protected void setFace( int v ){  
        face = v;  
    }  
    //...  
    public void lancer( ) { ... } }
```

setFace est protégée.

```
class DePipe extends De {  
    public void lancer( ){  
        setFace( facePipee );  
    }  
}
```

OK, car c'est une sous-classe.

```
class Joueur {  
    private De d = new De( );  
  
    public void doIt( ) {  
        d.setValeurFace( 3 );  
        d.lancer( );  
    }  
}
```

KO, Joueur n'est pas une sous-classe de De.

OK, public.

Chaînage de constructeurs

Les constructeurs ne s'héritent pas

On utilise `super` pour invoquer le constructeur de la super-classe

```
class DePipe extends De {  
    public DePipe( int fp ) {  
        super( );  
        setFacePipee( fp );  
    }  
    public DePipe( int fp,int startFace ) {  
        super( startFace );  
        setFacePipee( fp );  
    }  
    // ...  
}
```

Appelle explicitement le constructeur par défaut de la super-classe.

super doit apparaître à la première ligne du constructeur.

Les constructeurs sans instruction *super* invoquent implicitement le constructeur par défaut de la super-classe.

Passage de paramètres au constructeur de la super-classe grâce à l'instruction *super*.

Héritage et appels de méthodes

Les sous-classes peuvent invoquer les méthodes définies à la fois dans la super-classe (sauf privées) et dans la sous-classe

```
DePipe dp = new DePipe( 3 );  
int i = dp.getValeurFace ( );  
dp.lancer( );  
dp.setFacePipee( 5 );
```

Les références à la super-classe ne peuvent invoquer que les méthodes définies dans la super-classe

```
De d = new DePipe ( );  
int j = d.getValeurFace( );  
d.lancer( );  
d.setFacePipee( 5 );
```

//erreur de compilation

Erreur. Pourquoi ?

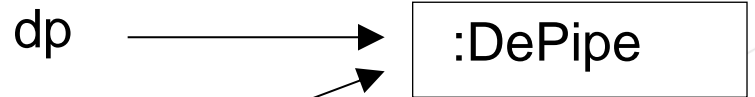
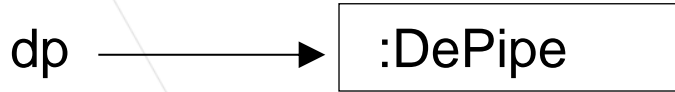
Références aux sous-classes

Une variable de super-classe peut référencer une sous-classe

Une sous-classe peut faire tout ce que font ses super-classes

```
DePipe dp = new DePipe( 3 );  
De d = null;  
d = dp;
```

OK, un DePipe est un De.



Messages polymorphes

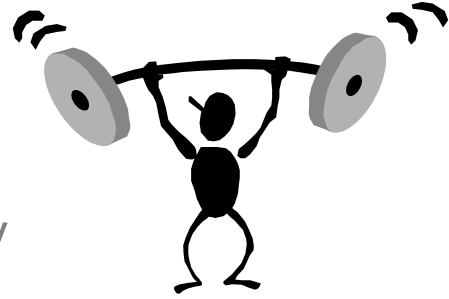
```
De[] tabDes = new De[2];  
tabDes[0] = new De();  
tabDes[1] = new DePipe( 3 );
```

```
for(De de : tabDes){  
    de.lancer( );  
}
```

Quelle est la méthode de
classe invoquée ?

Exercice

Construisez le cycle 2 du développement du Monopoly



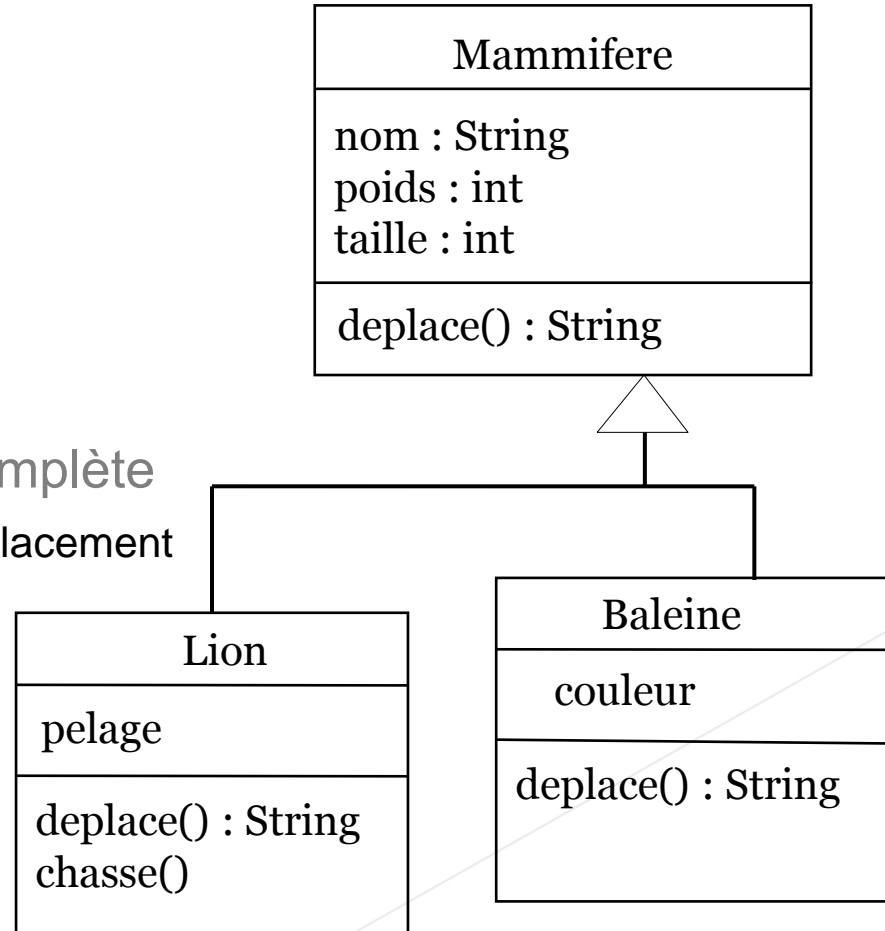
Classes abstraites

Les classes abstraites

- ne sont pas complètes
- ne doivent pas être instanciées
- définissent une interface pour toutes les sous-classes

La classe Mammifere n'est pas complète

- Comment pourrait-elle connaître le déplacement sans connaître le type du mammifère?



Codage d'une classe abstraite

```
public abstract class Mammifere{  
    private String nom;  
    private int poids, taille;  
    public abstract String deplace( );  
    public String getNom( ) { return nom; }  
    public int getPoids( ) { return poids; }  
    public int getTaille( ) { return taille; } }  
  
public class Baleine extends Mammifere {  
    public void deplace( )  
    { return "nage"}  
    // ... }
```

Les méthodes
abstraites ne peuvent
pas avoir d'
implémentation.

Les sous-classes doivent implémenter toutes les méthodes abstraites
ou être déclarées comme abstraites

Utilisation des classes abstraites

Elles s'utilisent comme toutes les classes d'héritages

```
Lion l = new Lion( );  
l.getNom( );  
System.out.println(l.deplace( )); //court
```

```
Mammifere m;  
m = new Mammifere ( ); //Erreur, pourquoi ?  
m = l;           //OK, pourquoi ?  
m.deplace( );  
m.getNom( );  
m.chasse( );    //Erreur, pourquoi ?
```

Classes « final »

Les classes final ne peuvent être spécialisées

```
final class DePipe extends De {  
    //clip  
}
```

```
class DePipeMemoire extends DePipe {  
    //clip  
}
```

Erreur !

Les classes « final » empêchent les redéfinitions

Utile pour les classes de sécurité et les classes concrètes

Résumé

L'héritage est l'implémentation Java de la relation de généralisation-spécialisation

- Permet de regrouper les attributs, associations et comportements communs au sein d'une super-classe commune

Le polymorphisme permet de traiter les objets de différentes classes d'une façon générique afin de réduire le fardeau de la maintenance

Polymorphisme + implémentation de l'héritage et des interfaces produit une combinaison puissante

Les classes abstraites sont des super-classes incomplètes

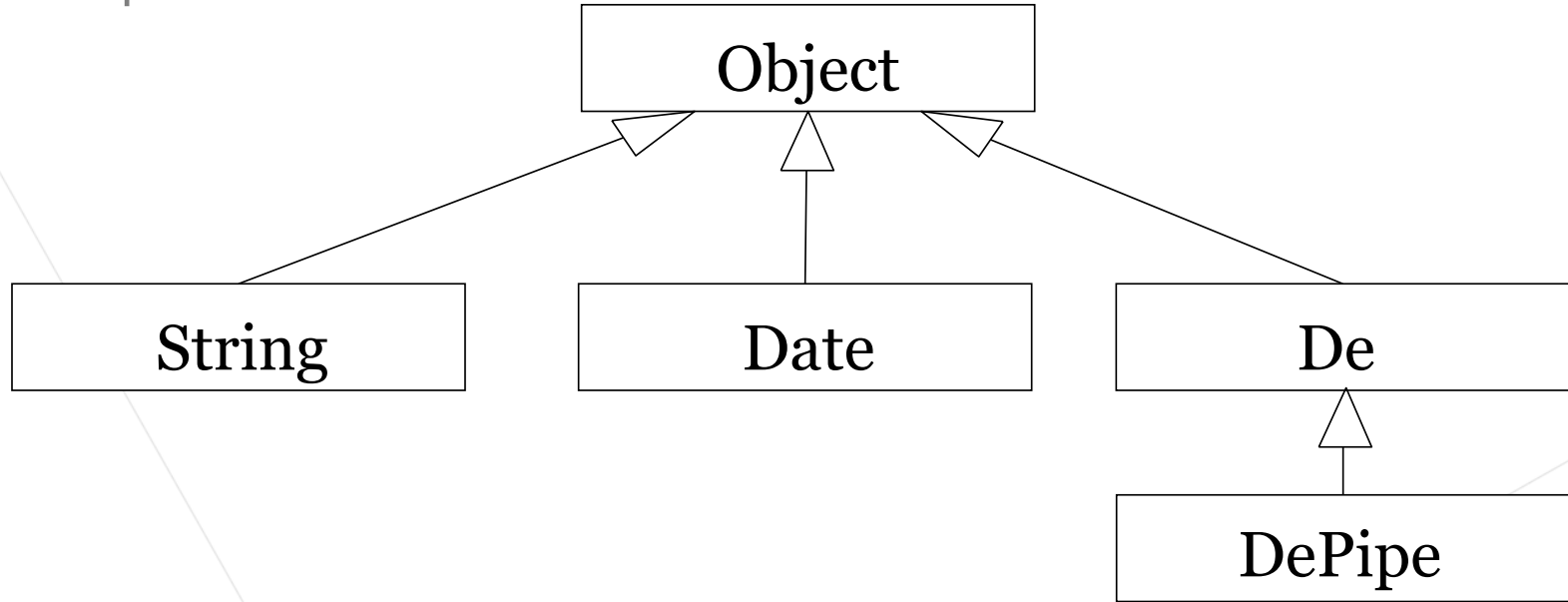
Java et la Conception Objet

La classe Object

Présentation

Object est une classe

Chaque classe en hérite directement ou indirectement



Méthodes de la classe Object

La classe Object définit des méthodes

```
public class Object {  
    public boolean equals( Object obj ) {...}  
    public String toString( ) {...}  
    // clip...  
}
```

Elle leur fournit une implémentation par défaut qui peut être redéfinie par les sous-classes

toString()

toString() permet de fournir une représentation sous forme de chaîne de l'objet

- Appelée implicitement par System.out.println() et « + »

```
System.out.println( obj );  
String s = "The object is " + obj;
```

- Par défaut, elle retourne le nom de la classe et la référence de l'instance
 - Redéfinissez-la avec une implémentation adaptée !

```
public class De {  
    public String toString( ) {  
        return "De avec valeurFace = " + getValeurFace( );  
    }  
}
```

equals()

Retourne vrai si les deux objets ont des valeurs égales

- par défaut equals compare les références d'instances à l'aide de ==
 - À redéfinir par chaque classe pour un comportement approprié

```
public class De {  
    public boolean equals( Object obj ) {  
        if ( obj instanceof De ) {  
            De other = (De) obj;  
            return this.getValeurFace( ) == other.getValeurFace( );  
        }  
        return false;  
    }  
}
```


Utilisation equals()

Utilisez equals() pour effectuer une comparaison de valeurs au lieu d'une comparaison d'identités

```
De de1 = new De(3);  
De de2 = new De(3);  
  
if ( de1 == de2 ) {...}    // toujours faux  
  
if ( de1.equals(de2) ) {...} // renvoie "true"
```

Résumé

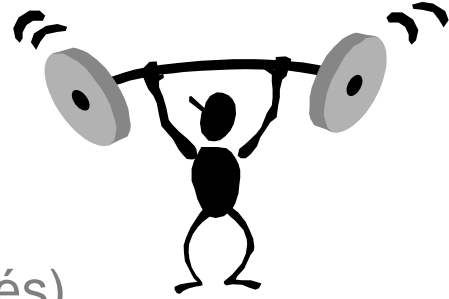
La classe Object est la super-classe de toutes les autres classes

Plusieurs méthodes sont héritées de la classe Object

- toString(): permet d'obtenir une représentation sous forme de chaîne d'un objet
- equals(): à redéfinir pour fournir un test d'égalité de valeur
- et bien d'autres...

Exercice

Procédez à l'analyse, à la conception et à la construction du cycle 3 du Monopoly (classes Propriétés)



Java et la Conception Objet

Les interfaces

Vous allez apprendre à...

Utiliser les interfaces Java

Mettre en œuvre l'héritage d'interfaces

Implémenter une interface

Définition

Définition

Quand utiliser les interfaces ?

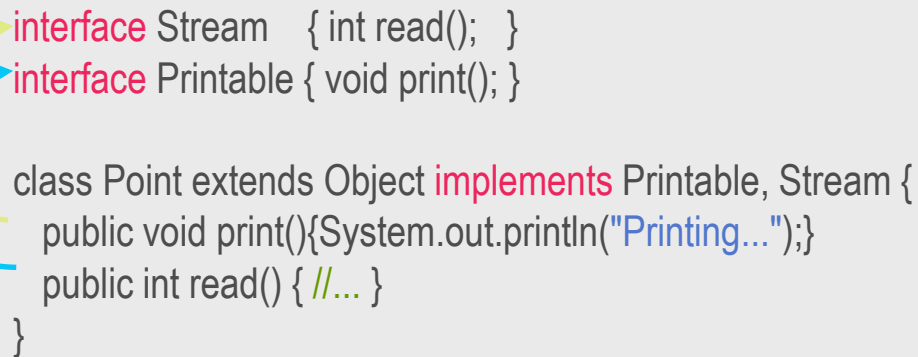
Interface

Une interface est un ensemble de signatures de méthodes et d'attributs

- Les méthodes sont **public** et **abstract**
- Les attributs sont **public**, **static** et **final**

Une classe peut implémenter plusieurs interfaces

- Elle doit fournir une implémentation pour chaque méthode
- Les méthodes doivent être publiques



```
interface Stream { int read(); }  
interface Printable { void print(); }  
  
class Point extends Object implements Printable, Stream {  
    public void print(){System.out.println("Printing...");}  
    public int read() { //... }  
}
```

Héritage multiple d'interfaces

Une interface peut hériter d'une ou de plusieurs autres interfaces

- L'héritage multiple d'interfaces est supporté par Java

```
interface Printable { void print(); }

interface Stream { int read(); }

interface DataStream extends Stream, Printable {
    float readFloat();}

class MyStream implements DataStream {
    public int    read() { ... }
    public float  readFloat() { ... }
    public void   print(){ ... }}
```


Types et interfaces

Une interface définit un nouveau type.

- Il est possible d'avoir une référence sur un objet implémentant une interface

Des objets complètement différents peuvent donc répondre au même message, à condition qu'ils implémentent la même interface.

L'opérateur `instanceof` peut être utilisé pour savoir si un objet implémente une interface donnée.

```
Point point = new Point();  
if ( point instanceof Printable) {  
    Printable p = (Printable)point;  
}
```

Exemple

```
interface Persistent {
    void save();
    void load();
}

class Client extends Personne implements Persistent {
    private int    numero;
    private String nom ;
    public void save() { // enregistrer dans un SGBD }
    public void load() { // charger à partir d'un SGBD }
}

class PersistentManager {
    private Persistent[] toManage ;
    //...
    void saveObjects() {
        for (int i = 0; i < toManage.length; i++)
            toManage[i].save();
    }
}
```

Interface générique

Une interface peut être utilisée pour définir un ensemble de méthodes à appliquer à des objets sans connaître leurs types

- Utilisation pour cela de la notion de générique **<T>**
- Le type sera précisé par la classe d'implémentation

```
interface DAO <T> {  
    int save(T data);  
    T findById(int id);  
}  
  
public class ClientDAO implements DAO <Client> {...}
```

Variables d'interfaces

Les variables déclarées dans une interface sont des constantes.

- Les mots-clés public, static et final ne sont pas nécessaires.

Les variables des interfaces doivent obligatoirement être initialisées.

```
interface MyInterface {  
    int MAX = 100;  
    int MIN = 0;  
}  
  
int i = MyInterface.MAX;
```

Quand utiliser les interfaces ?

Définition

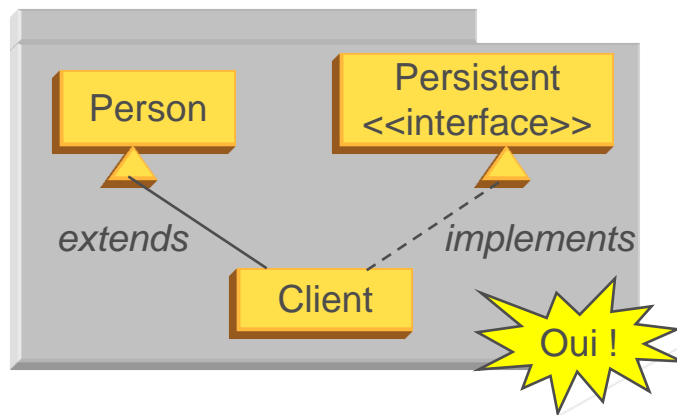
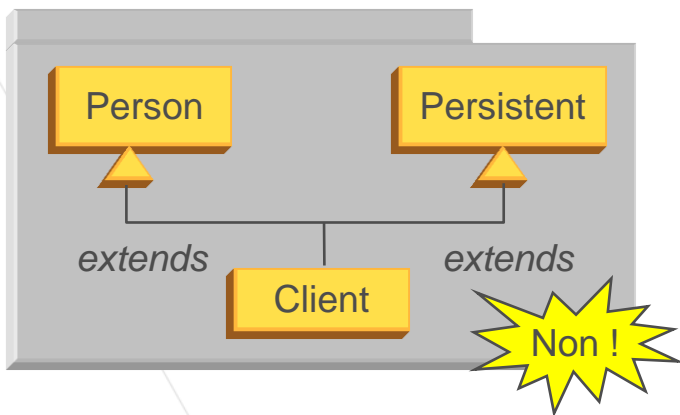
Quand utiliser les interfaces ?

Les services techniques

Certaines classes doivent prendre en charge des services techniques.

- Imprimer, Sauver...

Java ne supporte pas l'héritage multiple.



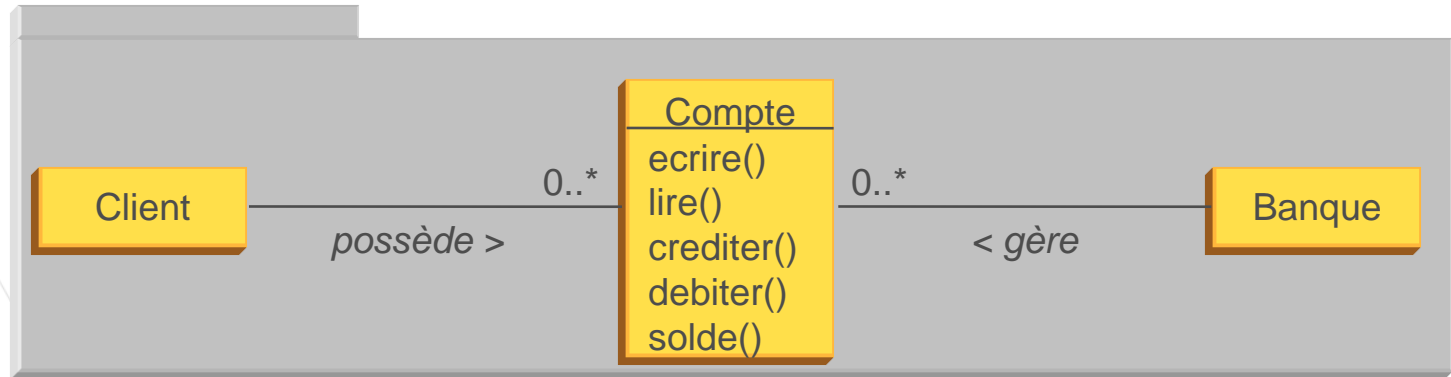
Utilisez les interfaces !

Conception par contrat 1/2

Définissez une interface lorsqu'une classe interagit avec une autre.

- Peut correspondre à des vues différentes sur un même objet,
- Une interface représente le contrat.

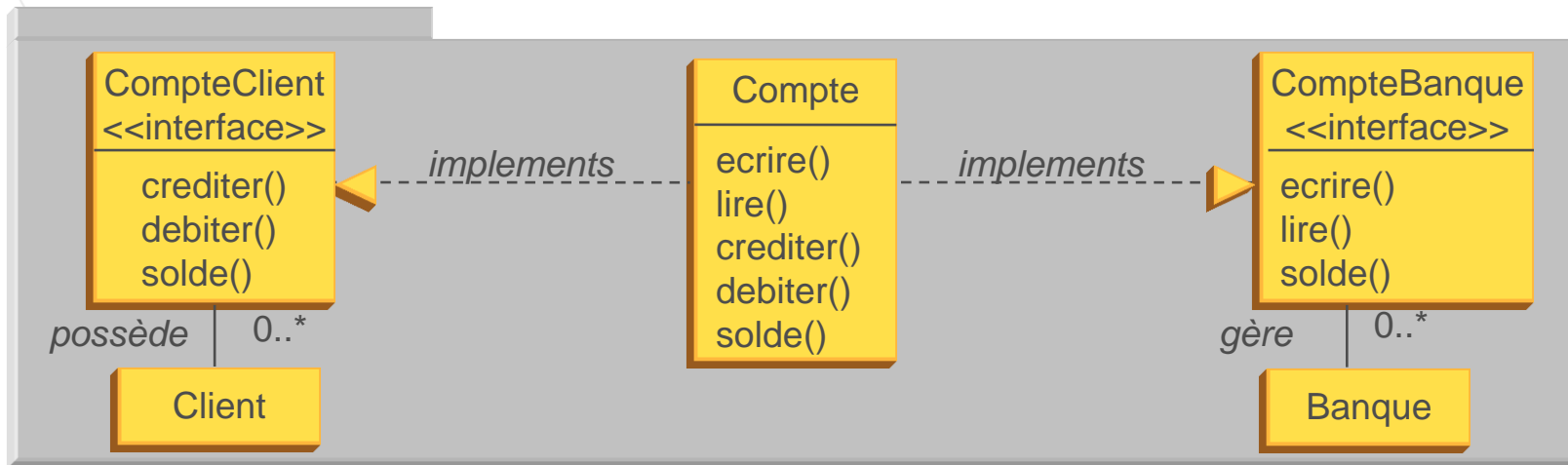
Exemple :



Conception par contrat 2/2

Le Client et la Banque ont des points de vue différents sur un Compte :

- Les méthodes utilisées ne sont pas les mêmes,
- Le Compte remplit deux contrats : un pour le Client (CompteClient), et un autre pour la Banque (CompteBanque).



Points clés

Une interface Java contient :

- la définition de méthodes publiques
- des constantes

Une classe peut implémenter une ou plusieurs interfaces

Les interfaces acceptent l'héritage multiple

Une interface représente un contrat entre deux classes

Java et la Conception Objet

Les collections Java 2

Présentation

Bien qu'il soit possible de créer des tableaux d'objets en Java, ce n'est pas une pratique courante

On utilise plutôt des collections (semblables à des tableaux) pour stocker des groupes d'objets

Java 2 propose un cadre de collections très sophistiqué

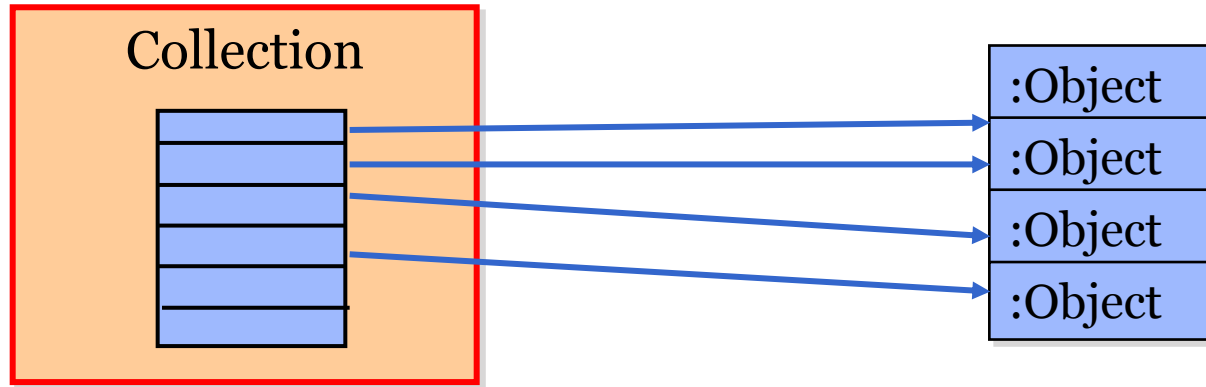
Trois grands types de collections existent :

- Les listes
- Les maps
- Les ensembles (sets)

Les collections stockent de multiples objets

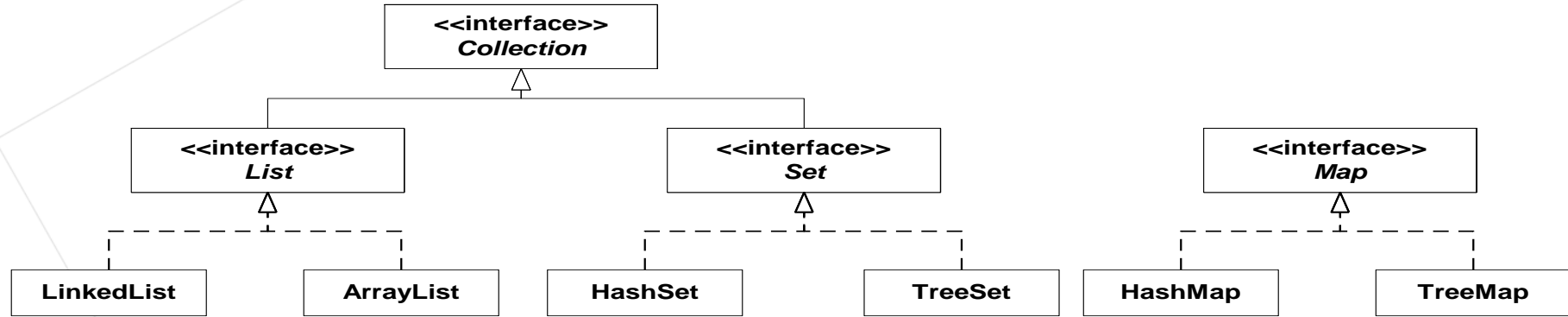
De l'intérieur, imaginez une collection comme une sorte de tableau

- Dans les faits, certaines collections sont implémentées de cette façon



- Les tableaux sont déclarés avec une taille de stockage fixe
- Si le stockage interne d'une collection est « plein », celle-ci augmentera de façon automatique et transparente sa capacité de stockage

Types de collection Java 2



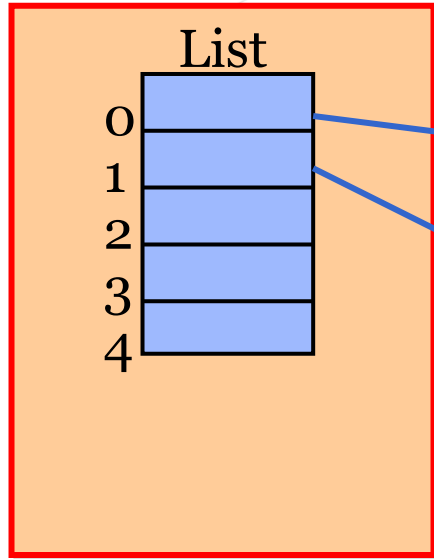
Situés dans le package java.util

Quelle différence y a-t-il entre les types List, Set et Map ?

- Les types « List » sont ordonnés et permettent l'indexation des éléments
- Les types « Set » ne permettent pas la duplication des éléments
- Les types « Map » associent les éléments avec des valeurs clés

Les collections sont typées pour éviter les casts inutiles et pour valider à la compilation les éléments qu'elles contiennent.

Les listes stockent les objets en ordre



Fortement typé

Opérateur diamant (Java 7)

```
List<Joueur> list = new ArrayList<>();
```

· · add() annexe l'objet à la fin du tableau.

```
list.add(new Joueur ("Cheval"));
```

```
list.add(new String("Voiture"));
```

```
list.add(new Joueur ("Canon"));
```

...

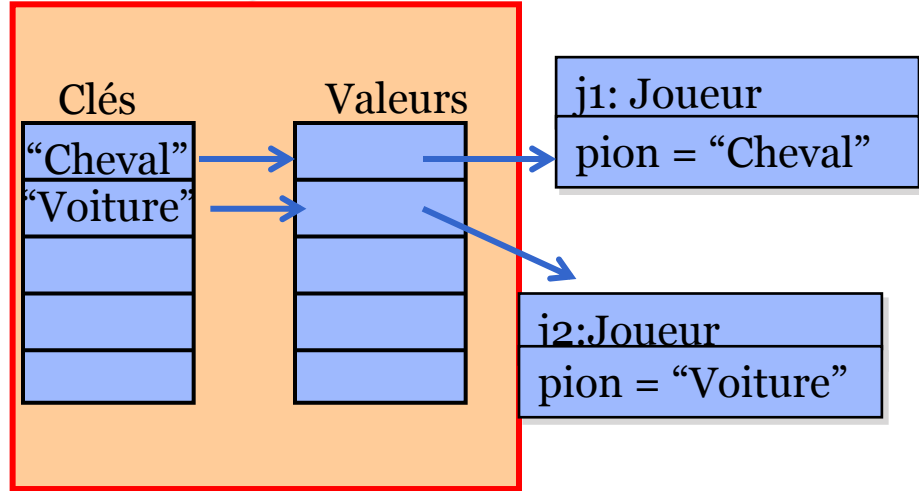
```
Joueur jo = list.get(0);
```

```
System.out.println(jo);
```

Error de
compilation
. Pourquoi ?

Imprime « Cheval ».

Les maps indexent les objets par une clé arbitraire



Une Map stocke des paires clé/valeur, clé et valeur étant des objets.

(Les clés sont souvent des entiers ou des chaînes)

```
Map<String,Joueur> map =  
    new HashMap<>();
```

```
...
```

```
Joueur j1, j2;
```

```
j1 = new Joueur("Cheval");
```

```
map.put("Cheval",j1);
```

```
j2 = new Joueur("Voiture");
```

```
map.put("Voiture",j2);
```

```
...
```

```
Joueur jo = map.get("Voiture");
```

```
System.out.println(jo);
```

Imprime « Joueur Voiture ».

Opérations des collections

List

- Insertion avec `add()`
- Récupération avec `get()`
- Clé par entier

et aussi

- `remove(Object)`
- `size()`
- `clear()`
- `contains(Object)`
- `isEmpty()`
- `toArray()`, ...

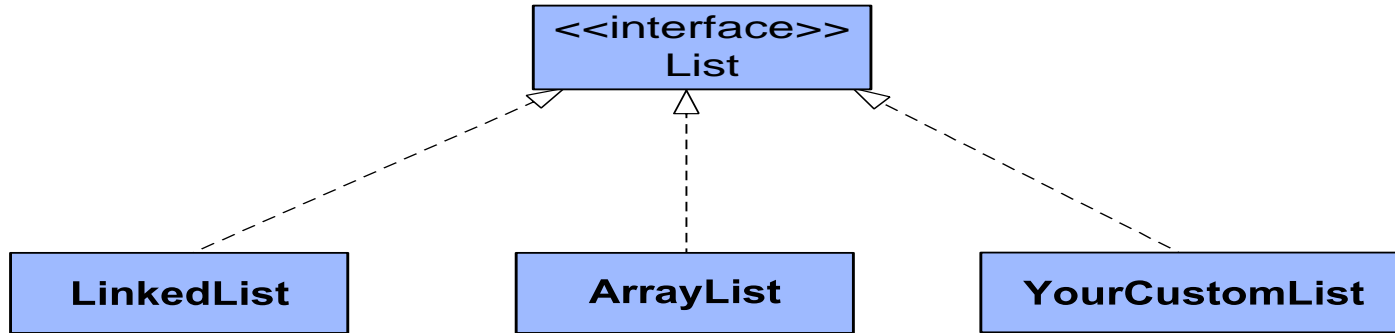
Map

- Insertion avec `put()`
- Récupération avec `get()`
- Clé d'objet arbitraire

et aussi

- `remove(Object)`
- `size()`
- `clear()`
- `containsKey(Object)`, `containsValue(Object)`
- `isEmpty()`
- `keySet()`, `values()`, ...

List, Set et Map sont des interfaces



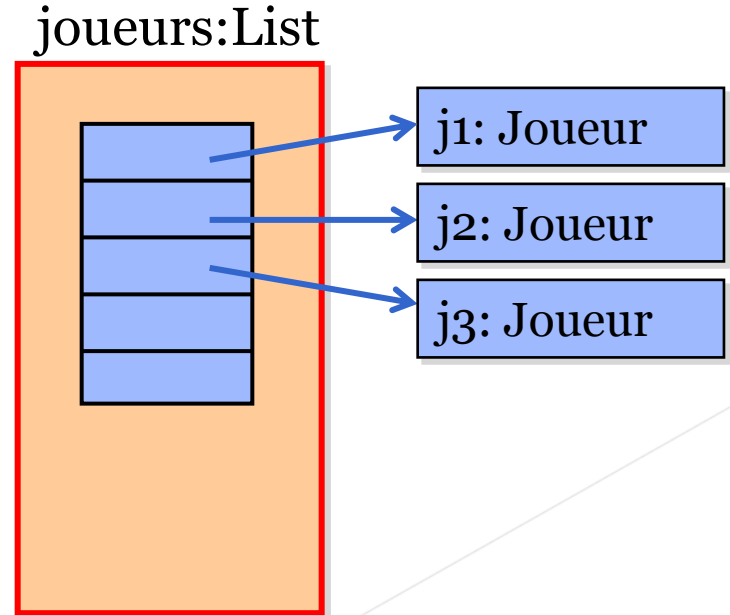
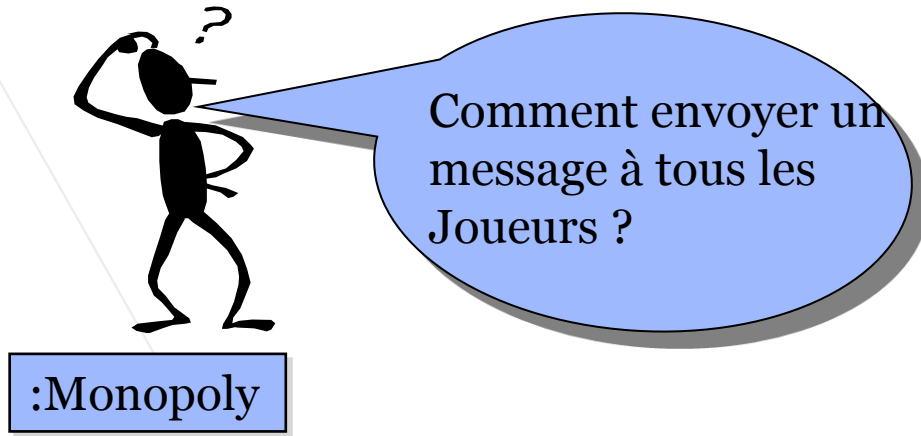
Si votre programme est entièrement écrit en termes de références à une interface, vous pouvez y connecter différentes implémentations sans rompre le code !

- `List<T> list = new ArrayList<T>();` ou
- `List<T> list = new LinkedList<T>();`

rien d'autre ne casse !

Accéder à tous les éléments d'une collection

On souhaite accéder à chaque membre de la collection une seule et unique fois



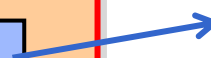
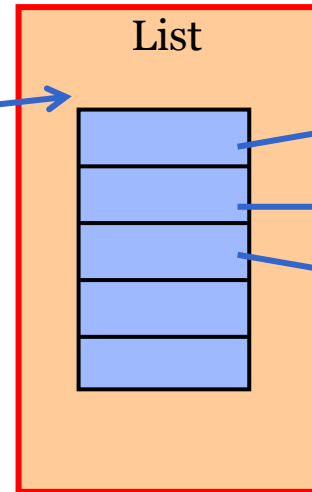
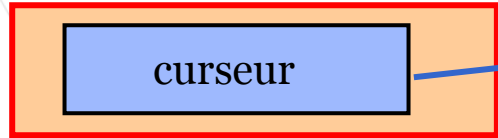
L'itération

Moyen standard et contrôlé d'accéder à chaque élément de la collection

Les itérateurs fonctionnent comme curseur sur une collection

- Ils ont connaissance de la structure interne de la collection

Itérateur



o1:Objet



o2:Objet



o3:Objet

Iterator est une interface

Chaque collection peut produire un itérateur qui parcourt la collection

- `Iterator<Type> = list.iterator();`

L'itérateur est une interface du package `java.util`

<code><<interface>></code> <code>Iterator<T></code>
<code>next():T</code> <code>hasNext():boolean</code> <code>remove()</code>

Parcours d'une collection via itérateur

Au sein d'une boucle

```
Iterator<String> it = myList.iterator();  
while( it.hasNext() ) {  
    String myString = it.next();  
    ...  
}
```

Garantit que l'on parcourt chaque élément une seule et unique fois

L'itération sur un Map est différente. Pourquoi ?

L'itération des Maps est plus complexe

Les Maps peuvent retourner l'ensemble de leurs clés en tant que Set, et de leurs valeurs en tant que Collection

Il est possible d'obtenir un itérateur sur chacun de ces éléments :

<code><<interface>> Map<K,V></code>
<code>entrySet():Set keySet():Set<K> values():Collection<V></code>

```
Set keySet<K> = map.keySet();  
Iterator<K> it = keySet.iterator();  
// now iterate on the Set
```

Résumé

Les collections sont le mode standard de stockage d'objets multiples

Les « List », « Map » et « Set » sont les principaux types de collection

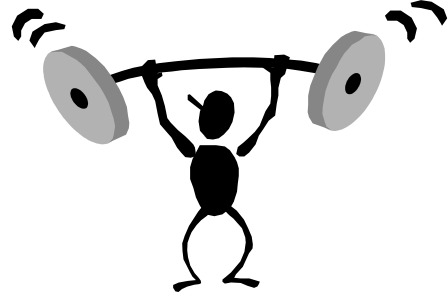
Java 5 a introduit de nouvelles collections « typées » dites génériques

L'itération permet de parcourir tous les éléments d'une collection

On s'efforcera de manipuler les collections à travers leurs interfaces

Exercice

Modifiez votre code de façon à utiliser des collections selon les instructions de votre formateur



Java et la Conception Objet

Méthodes et variables statiques

Présentation

Les membres statiques sont des membres définis et portés par la classe et non par ses instances

Les données (attributs) comme les méthodes peuvent être statiques

Les données statiques

Les variables statiques :

- ont la durée de vie du programme qui les crée
- sont partagées par toutes les instances de la classe

```
class Compteur {  
    private static int compte = 0;  
    public static int getCompte( ){ return compte;}  
    public static void setCompte( int c ){ compte = c; }  
}
```

Les variables statiques sont stockées dans l'espace mémoire occupé par la classe, et non dans celui d'une instance particulière.

Méthodes statiques

Les méthodes statiques ont une portée globale

```
public void doIt( ) {
```

```
    Compteur c;
```

```
    Compteur.setCompte( 10 );  
    int i = Compteur.getCompte( );
```

```
    c.setCompte( 20 );  
    int k = c.getCompte( );
```

```
}
```

On les invoque
directement à
l'aide du nom
de la classe

On considère ici que
setCompte est déclarée
comme static

Les méthodes statiques n'ont pas de pointeur this

Visibilité - restrictions

Les membres statiques sont accessibles à partir des instances, mais la réciproque est fausse

Les méthodes statiques

- ne peuvent pas accéder à des méthodes non statiques ou à des variables d'instances non statiques
- ne peuvent pas être redéfinies en méthodes non statiques dans les classes dérivées

Exemple

```
class Compteur {  
    private static int compte = 0;  
    private int n;  
    public static void setCompte( int c ) {  
        compte = c;  
        doIt( );  
        n = 100;  
    }  
    public void doIt( ) { ... }  
}
```

Erreur ! Pourquoi ?

Variables statiques finales publiques

Les variables finales sont généralement rendues publiques et statiques

```
class De {  
    public final static int MAX_LANCER = 6;  
}
```

On les utilise directement depuis le nom de la classe :

- `int s = De.MAX_LANCER;`

Résumé

Les membres statiques appartiennent à la classe, non à l'une de ses instances

Les membres statiques sont toujours disponibles

- Leur cycle de vie n'est lié à aucune instance

Exercice (Optionnel)



Design Pattern Singleton : Une unique instance de la classe

```
public class Singleton {  
    //Constructeur privé  
    //--> empêche de créer une instance depuis l'extérieur  
    private Singleton(){}   
  
    //attribut statique, partagé par toutes les instances  
    private static Singleton instance;  
  
    //Accesseur statique -- méthode appartient à la classe  
    public static Singleton getInstance() {  
        //Test si l'instance existe  
        if(instance == null){  
            //créer l'instance unique  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Modifiez le système en appliquant au gobelet le pattern Singleton, de manière à ne plus avoir à le passer en paramètre.



Oooops ...

Java et la Conception Objet

Gestion des exceptions

Le besoin

Le besoin

La spécification

La classification

Gestion des exceptions

À quoi servent les exceptions ?

Elles permettent de gérer toutes les situations « hors contrat » indépendamment du traitement nominal

```
// équation du second degré
class Equation {...}

// exemple:  $x^2 + 1 = 0$  (pas de solution réelle)
Equation eq = new Equation( 1, 0, 1 );
double resultat = eq.solution1();
System.out.println("Resultat = " + resultat );
```

Comment...

- Distinguer un résultat valide d'un code d'erreur ?
- Propager jusqu'au gestionnaire de ce cas exceptionnel les informations qui lui sont nécessaires (code d'erreur, données de contexte) ?

Exemple

```
try {  
    // exemple:  $x^2 + 1 = 0$  (pas de solution réelle)  
    Equation eq = new Equation( 1, 0, 1 );  
    double resultat = eq.solution1();  
  
    System.out.println("Resultat = " + resultat );  
}  
catch( ExceptionPasDeSolution p ) {  
    // Gestionnaire du cas exceptionnel  
    System.out.println("Pas de solution" );  
}
```

```
// cas exceptionnel à traiter séparément  
class ExceptionPasDeSolution extends Exception {}
```

```
// equation du second degré  
class Equation {...}  
    public Equation( double a, double b, double c );  
    private double delta() { return b*b - 4*a*c; }  
  
    public double solution1( )  
        throws ExceptionPasDeSolution {  
        double discr = delta( );  
  
        if( discr < 0 )  
            throw new ExceptionPasDeSolution();  
        return (b + Math.sqrt(discr) ) / (2*a);  
    }  
}
```



exception

La spécification

Le besoin

La spécification

La classification

Gestion des exceptions

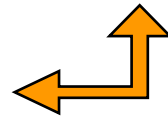
Java OBLIGE à gérer les exceptions

Vous DEVEZ spécifier les exceptions levées et non traitées dans la signature des opérations

```
public Equation( double a, double b, double c )  
{  
    private double delta() { return b*b - 4*a*c; }  
    public double solution1()  
        throws ExceptionPasDeSolution {...}  
}
```

```
void compute() {  
    try {  
        ...  
        eq.solution1();  
    } catch(ExceptionPasDeSolution pds) {  
        ...  
        return;  
    }  
}
```

```
void compute()  
    throws ExceptionPasDeSolution {  
        ...  
        eq.solution1();  
        ...  
}
```



Vous avez le choix !

throw et throws

Si un gestionnaire d'exceptions capte une exception mais ne sait pas la traiter, il peut la propager au gestionnaire suivant dans la pile des appels

```
void compute() throws ExceptionPasDeSolution {  
    try {  
        ...  
        eq.solution1();  
    } catch( ExceptionPasDeSolution pds ) {  
        ...  
        throw pds;  
    }  
}
```


Le bloc finally

finally permet de factoriser du code.

- Il est toujours exécuté, qu'une exception survienne ou non.

```
try {  
    // ouvrir un fichier  
    // lire et écrire des données...  
} catch( IOException i) {  
    // traiter l'exception  
} finally {  
    // fermer le fichier  
}  
...
```

**TOUJOURS
executé**

Les traces

Une exception contient des informations sur son contexte

- La pile dépile les appels amenant à la méthode qui a déclenché l'exception
 - Affichée via `printStackTrace()`

```
try { ...  
}  
catch( IOException ex) {  
    System.out.println("Erreur de type " + ex);  
    ex.printStackTrace();  
}
```

- Optionnellement un message
 - L'un des constructeurs de la classe `Throwable` prend une `String` en paramètre.
 - Permet d'enrichir les traces avec un message obtenu par `getMessage()`.

La classification

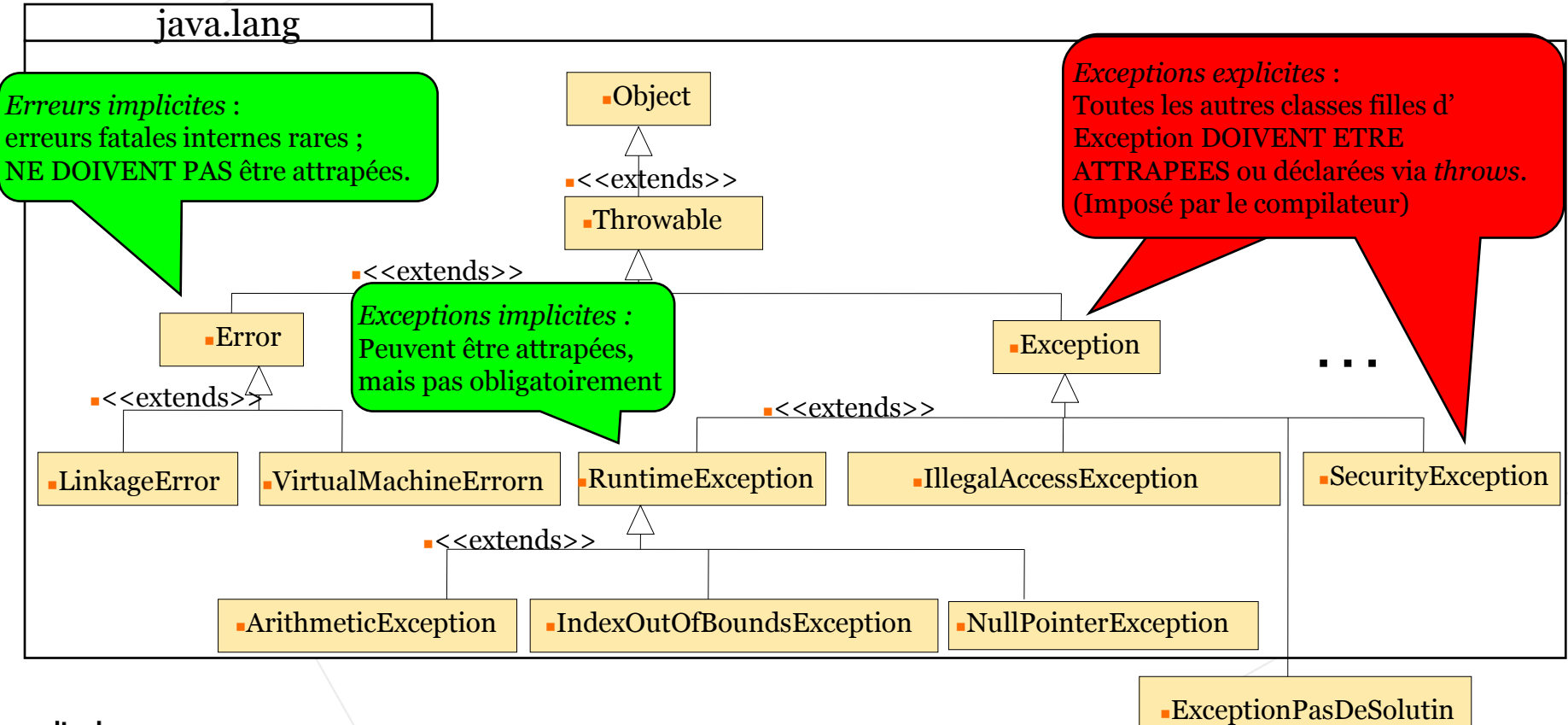
Le besoin

La spécification

La classification

Gestion des exceptions

Classification des exceptions



Gestion des exceptions

Le besoin

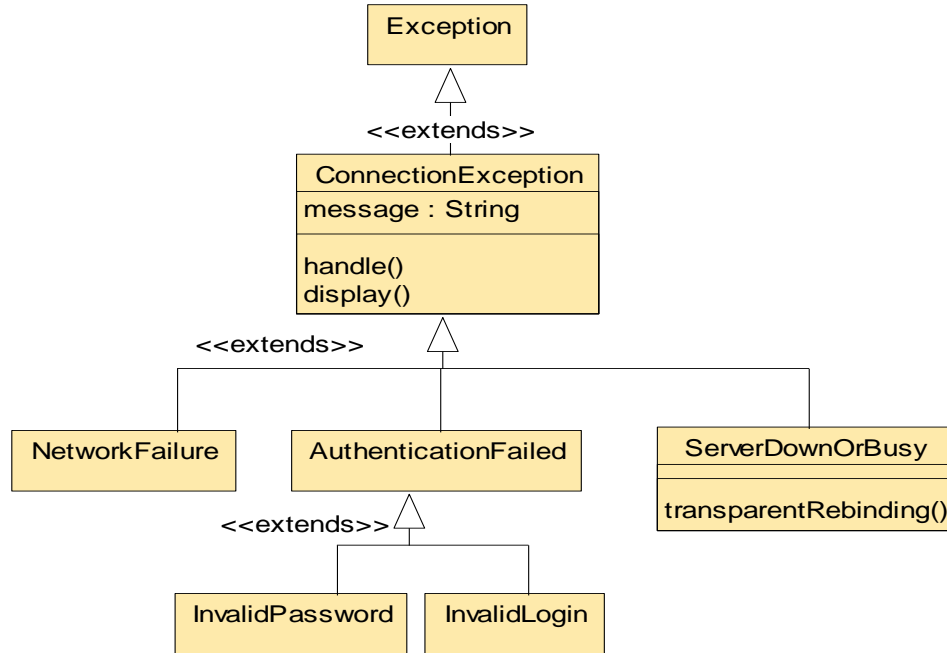
La spécification

La classification

Gestion des exceptions

Exceptions et polymorphisme

com.valtech.exceptions



Dans notre exemple, `handle()` & `display()` sont redéfinies dans les sous classes

Exceptions et polymorphisme

Construire un gestionnaire polymorphe et sélectif.

```
try {  
    monServer.connect();  
}  
catch( ServerDownOrBusy e){  
    // traitement spécifique  
    e.transparentRebinding();  
    e.handle();  
    e.display();}  
catch (ConnectionException e){  
    //traitement polymorphe  
    e.display();  
    e.handle();}
```

Que se passe-t-il si l'on inverse l'ordre des deux blocs catch ???

Conseils & astuces

N'abusez pas du temps d'exécution système (plus que substantiel) nécessaire à la levée des exceptions

Ne procédez pas à une « micro-gestion » de chaque ligne de code susceptible de lever une exception

- Utilisez un seul bloc « try » avec plusieurs blocs « catch »

N'ignorez pas les exceptions !

- On ne veut JAMAIS voir : `catch (Exception e) { }`
- `printStackTrace()` est un minimum.

Résumé

Il existe des exceptions explicites et implicites

Les exceptions sont lancées par throw

Et gérées par les clauses try/catch/finally

Un bloc try peut avoir des blocs catch multiples

Les exceptions explicites non attrapées dans une méthode doivent être déclarées par celle-ci avec throws

Exercice (Optionnel)

Procédez à l'analyse, à la conception et à la construction du cycle 4 du Monopoly (Exceptions)

