



Apprendre les concepts
objets avec Java
Cahier de TP

Copyright - OXIANE, 98 avenue du Gal Leclerc, 92100 Boulogne. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de OXIANE et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de OXIANE.

OXIANE, le logo OXIANE sont des marques de fabrique ou des marques déposées, ou marques de service, de OXIANE en France et dans d'autres pays.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REpondre A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Copyright - OXIANE, 98 avenue du Gal Leclerc, 92100 Boulogne. All rights reserved .

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of OXIANE and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from OXIANE suppliers.

OXIANE, the OXIANE logo are trademarks, registered trademarks, or service marks of OXIANE in France and other countries.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

TP 1 : fonctions et bibliothèque de fonctions

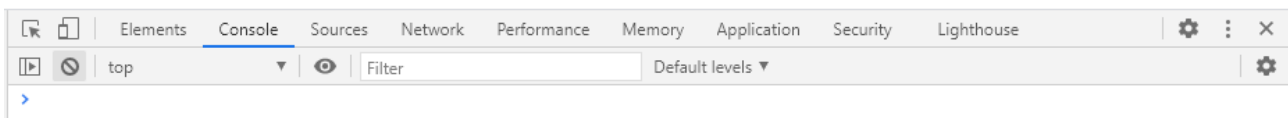
Les exercices suivants sont très scriptés afin que chacun puisse les dérouler sans être bloqué. Nous attirons l'attention sur le danger d'une telle facilité. Le but des exercices n'est pas de vérifier qu'ils fonctionnent bien. Vous devez, bien entendu, prendre le temps de comprendre les instructions proposées pour les assimiler et ainsi acquérir de nouvelles compétences.

Exercice 1 : notion de fonction et de bibliothèque de fonctions

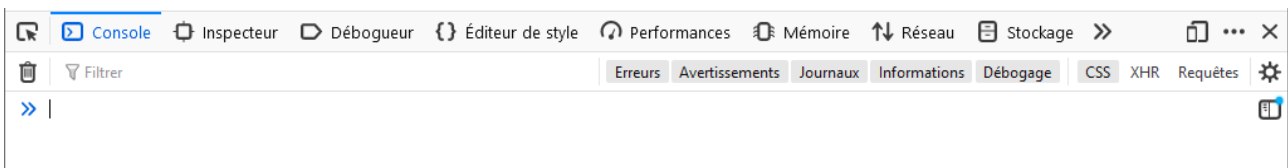
Dans cet exercice, nous illustrons la définition de fonction et d'appel de fonction. Pour simplifier, nous utiliserons le langage JavaScript à l'intérieur d'un navigateur tel que Chrome ou Firefox.

- > Ouvrir un navigateur Chrome ou Firefox et basculer en mode **console**
 - pour cela, utiliser la combinaison de touche **Ctrl+Maj+i**
 - puis sélectionner l'onglet **Console**

Chrome



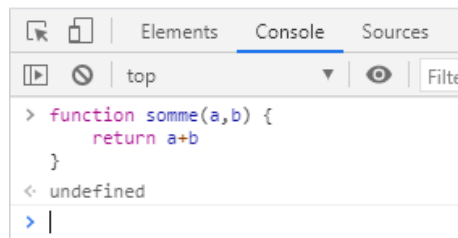
Firefox



- > définir une fonction **somme(a, b)** additionnant deux nombres
 - la syntaxe est la suivante (noter que JavaScript n'est pas typé)

```
function somme(a,b) {
    return a+b
}
```

- la fonction effectue l'addition (+) entre les deux valeurs qu'elle reçoit en paramètre
- puis retourne le résultat
- noter que l'interpréteur **JavaScript** affiche toujours un résultat (ici **undefined**) pour confirmer



- > Invoker la fonction **somme** avec les paramètres **2** et **3** et vérifier le bon résultat (**5**)
 - il suffit d'utiliser le nom de la fonction et d'utiliser les parenthèses pour spécifier les arguments

```
somme(2, 3)
```

```
> somme(2,3)
< 5
```

> Invoquer la fonction **Math.random()**

- noter que cette fonction est préfixée **Math**
- il s'agit en quelque sorte d'une bibliothèque regroupant les fonctions mathématiques
- noter que cette fonction retourne un nombre de l'intervalle **[0,1[**

```
> Math.random()
< 0.15253564227881777
>
```

- ainsi, il suffit de **multiplier** par **6** pour obtenir un nombre dans l'intervalle **[0, 6[**

```
Math.random() * 6
```

```
> Math.random() * 6
< 1.917968267401886
>
```

> Utiliser la fonction **Math.trunc()** pour convertir en un entier entre **0** et **5**

```
Math.trunc(Math.random() * 6)
```

```
> Math.trunc(Math.random() * 6)
< 4
> |
```

> Ajouter **1** pour obtenir un entier entre **1** et **6**

```
Math.trunc(Math.random() * 6) + 1
```

```
> Math.trunc(Math.random() * 6) + 1
< 6
> |
```

Exercice 2 : créer une fonction *alea()*

> Créer une fonction **alea()** retournant un entier aléatoire entre **1** et **6**

- essayer de l'écrire seul en s'inspirant de la définition de la fonction **somme()** plus haut
- vérifier son bon fonctionnement
- la solution se trouve sur la page suivante

```
> function alea() {
    return Math.trunc(Math.random() * 6 + 1)
}
< undefined
> alea()
< 1
> alea()
< 2
> alea()
< 2
> alea()
< 6
> |
```

Exercice 3 : modifier la fonction alea() pour la paramétrer

L'univers des jeux de plateau utilise abondamment toutes sortes de dés à jouer à plus de 6 faces. Redéfinir la fonction **alea()** pour prendre en compte cette modification.

- > Expérimenter une solution à base d'une variable globale appelée **max**
 - rappeler l'historique de la saisie avec la flèche vers le haut et modifier la définition

```
function alea() {
    return Math.trunc(Math.random() * max + 1)
}
```

- vérifier le comportement en invoquant cette nouvelle fonction **alea()**

```
> function alea() {
    return Math.trunc(Math.random() * max + 1)
}
< undefined
> alea()
✖ ▶ Uncaught ReferenceError: max is not defined
    at alea (<anonymous>:2:38)
    at <anonymous>:1:1
> |
```

- la nouvelle définition de la fonction est prise en compte mais la variable **max** n'est pas définie
- il faut renseigner la variable avant de faire l'appel de la fonction

```
> max = 20
< 20
> alea()
< 19
> |
```

- **attention** : ces exemples sont purement illustratifs et ne respectent pas les bonnes pratiques
 - noter que l'appel complexe n'est pas la seule faiblesse de cette solution
 - il faut connaître à l'avance le nom de cette variable **max** et on peut risquer de l'utiliser ailleurs
- > Expérimenter la solution à base de paramètre

- modifier la définition de la fonction pour introduire le **paramètre max**

```
function alea(max) {
  return Math.trunc(Math.random() * max + 1)
}
```

- tester

```
> function alea(max) {
  return Math.trunc(Math.random() * max + 1)
}
< undefined
> alea(100)
< 9
> alea(100)
< 91
> |
```

- noter que la variable **max** existe toujours au niveau global (notion de portée, scope en anglais)

```
> max
< 20
>
```

- elle peut être supprimée avec l'instruction **delete max**

```
> delete max
< true
> max
✖ ▶ Uncaught ReferenceError: max is not defined
   at <anonymous>:1:1
```

- noter que la fonction **alea()** exige maintenant un paramètre

```
> alea(6)
< 5
> alea()
< NaN
> |
```

- **NaN** (Not a Number) est une valeur spéciale obtenue lors d'un calcul erroné
- cette expérience nous montre qu'il ne peut pas y avoir deux fonctions **alea()** dans le système
- c'est une contrainte importante des langages classiques : les fonctions sont uniques

TP 2 : objet, encapsulation, état

La programmation orientée objet est avant tout une manière de concevoir en gagnant en abstraction, en s'éloignant des complexités techniques de l'informatique.

- Quelles sont les choses qu'on veut représenter ?
- Quelles sont les caractéristiques de ces choses ?
- Quels services nous apportent ces choses ?

Dans la vie courante ces choses sont des **objets**, les objets que l'on manipule comme la télévision et sa télécommande, le grille pain, l'ordinateur, la feuille de papier représentant une facture, etc.

Ces objets fabriqués sont généralement « encapsulés » : ils présentent un boîtier pour protéger l'intérieur contre la curiosité et surtout les mauvaises utilisations. La télévision et sa télécommande présentent toutes les deux un boîtier pour masquer tous les composants électroniques internes.

Nous essayons de simuler des dés à jouer. Certains dés ont 6 faces, d'autres 20, c'est une caractéristique propre au dé. Ces dés peuvent rouler et ils s'arrêtent sur une face au hasard, c'est la **valeur** du dé. Chaque dé a évidemment sa valeur. Tant que le dé ne roule pas, il reste sur la même valeur (état du dé).

> Créer un objet représentant un dé à **6 faces** et le référencer dans une variable appelée **unDé**

- la syntaxe JavaScript permet de créer des objets de manière très simple

```
unDé = { nbFaces: 6 }
```

```
> unDé = { nbFaces: 6 }  
< ▶ {nbFaces: 6}
```

- noter l'affichage particulier de l'objet entre accolades

> Consulter la propriété **nbFaces** de l'objet

- la syntaxe utilise la notation pointée **variable.propriété**

```
unDé.nbFaces
```

```
> unDé.nbFaces  
< 6
```

- noter que JavaScript ne respecte pas tout à fait l'encapsulation ici

> Essayer d'obtenir la valeur du dé

```
unDé.valeur
```

```
> unDé.valeur  
< undefined
```

- la propriété **valeur** n'est pas définie, on retrouve cette valeur **undefined** déjà aperçue plus tôt
- elle correspond à l'absence de valeur, dans la même idée que le **NULL** du langage SQL
- tous les langages objets proposent cette notion, généralement appelée **undefined**, **null** ou **nil**

> Noter qu'il est possible d'affecter une valeur après la création de l'objet

```
unDé.valeur = 5
unDé.valeur
```

```
> unDé.valeur = 5
< 5
> unDé.valeur
< 5
> |
```

- observer là encore le non respect de l'encapsulation
- la conséquence immédiate est qu'il nous serait possible d'affecter la valeur **7** à ce dé à **6** faces
- JavaScript est un langage permissif, c'est au développeur de respecter l'encapsulation
- **remarque** : il existe des moyens, non abordés ici, de créer des objets réellement encapsulés

> Créer un second dé avec **20 faces**, une **valeur** aléatoire, référencé par la variable **unSecondDé**

```
unSecondDé = { nbFaces: 20, valeur: alea(20) }
```

```
> unSecondDé = { nbFaces: 20, valeur: alea(20) }
< {nbFaces: 20, valeur: 16}
> |
```

> Consulter la valeur de chaque dé

```
unDé.valeur
unSecondDé.valeur
```

```
> unDé.valeur
< 5
> unSecondDé.valeur
< 16
> |
```

- observer que chaque dé a bien sa propre **valeur**, ce qui constitue son **état**

Exercice 2 : identité et référence

- noter que nous pouvons modifier cet état pour avoir deux dés avec la même valeur

```
unSecondDé.valeur = 5
```

```
> unSecondDé.valeur = 5
< 5
```

- nous avons maintenant deux dés **différents** qui ont la **même valeur**
- les deux objets sont bien différents comme nous le confirme l'opérateur **==**

```
unDé == unSecondDé
```

```
> unDé == unSecondDé
< false
```

- cet opérateur distingue l'**identité** des objets et non leur **égalité** (leur valeur)
- c'est plus facile de retenir que l'opérateur teste si les variables contiennent le même objet

- **remarque** : l'opérateur `==` de JavaScript est un peu plus subtile et complexe que ça
- **attention** : ne pas confondre `=` (affectation) et `==` (comparaison)

> Créer une 3^e variable **lePremierDé** référençant le premier dé **unDé**

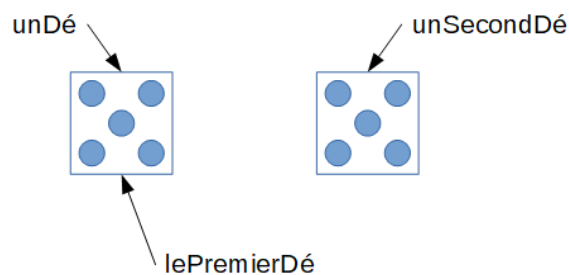
```
lePremierDé = unDé
```

```
> lePremierDé = unDé
< ▶ {nbFaces: 6, valeur: 5}
```

- noter qu'il s'agit seulement d'une autre **référence** (un autre nom) pour **unDé**

```
> lePremierDé == unDé
< true
```

- c'est une notion souvent difficile à comprendre au début, bien avoir en tête le schéma suivant :



important : les variables ne sont que des noms locaux donnés aux choses

> Vérifier que c'est bien compris en prédisant ce qu'il va se passer et s'afficher avant d'exécuter

```
lePremierDé.valeur = 2
lePremierDé.valeur      //quelle valeur va s'afficher ici ?
unSecondDé.valeur       //et ici ?
unDé.valeur              //et enfin ici ?
```

TP 3 : méthodes et communication par messages

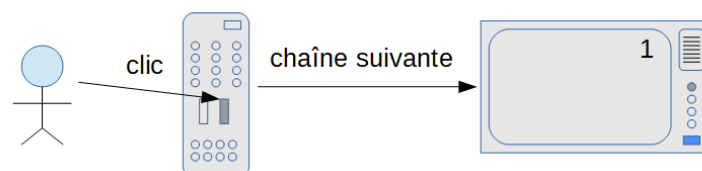
- > Ajouter une méthode **rouler()** à l'objet **unDé** (ou **lePremierDé** puisque c'est pareil)
 - cette méthode doit calculer une nouvelle valeur pour le dé, en utilisant la fonction **alea()**
 - la syntaxe est particulière à JavaScript : une méthode est une propriété contenant une fonction
 - noter que les valeurs propres à l'objet (ses propriétés) doivent être préfixées **this**.

```
unDé.rouler = function() { this.valeur = alea(this.nbFaces) }
```

- pour rouler, notre dé va calculer un entier aléatoire correspondant à **son** nombre de faces
- puis modifier **son** état (sa propriété **valeur**)

```
> unDé.rouler = function() { this.valeur = alea(this.nbFaces) }
< f () { this.valeur = alea(this.nbFaces) }
```

- > Activer cette opération
 - on demande à **unDé** de **rouler**, c'est-à-dire d'activer sa méthode **rouler()**
 - ça peut paraître peu naturel mais il s'agit de mettre en place un réflexe d'échange de message
 - penser à ce qu'il se passe lorsqu'on appuie sur un bouton de sa télécommande
 - on transmet un ordre à la télécommande « bascule sur la chaîne n° 3 » ou bien « plus fort »
 - la télécommande retraduit cet ordre en impulsions infrarouges (message) vers la télévision
 - cette dernière reçoit l'ordre et l'exécute



- pendant des années, on a rêvé de pouvoir le faire à la voix et c'est aujourd'hui possible
- on peut effectivement parler à sa télévision ou lui faire des gestes, c'est-à-dire communiquer
- penser à Toy Story ou aux tribus primitives : nous avons besoin de prêter une âme aux choses
- il suffit de retrouver son âme d'enfant : jouer à parler à des objets ou les faire parler entre eux
- vérifier ensuite la valeur du dé

```
unDé.rouler()
unDé.valeur
```

```
> unDé.rouler()
< undefined
> unDé.valeur
< 6
```

- **remarque** : la syntaxe JavaScript permet de combiner les deux instructions sur une seule ligne

```
unDé.rouler(); unDé.valeur
```

```
> unDé.rouler(); unDé.valeur
< 4
> unDé.rouler(); unDé.valeur
< 6
```

> Écrire à l'aide de l'objet **console**

- lui envoyer un message **log()** avec une chaîne de caractères **"Hello"** en paramètre

```
console.log("Hello")
```

```
> console.log("Hello")
Hello
< undefined
```

> Modifier la méthode **rouler()** pour afficher la nouvelle valeur dans la console puis tester

```
unDé.rouler = function() {
  console.log("Le dé roule...")
  this.valeur = alea(this.nbFaces)
  console.log("...et s'arrête sur le " + this.valeur)
}
unDé.rouler()
```

```
> unDé.rouler = function() {
  console.log("Le dé roule...")
  this.valeur = alea(this.nbFaces)
  console.log("...et s'arrête sur le " + this.valeur)
}
< f () {
  console.log("Le dé roule...")
  this.valeur = alea(this.nbFaces)
  console.log("...et s'arrête sur le " + this.valeur)
}
> unDé.rouler()
Le dé roule...
...et s'arrête sur le 4
< undefined
```

> Copier la méthode dans **unSecondDé** et tester

```
unSecondDé.rouler = function() {
  console.log("Le dé roule...")
  this.valeur = alea(this.nbFaces)
  console.log("...et s'arrête sur le " + this.valeur)
}
unSecondDé.rouler()
```

> faire rouler **unDé** et/ou **unSecondDé** de manière à obtenir un **6** et un **20**

- arriverez-vous à faire mieux que mes 3 coups ? (véridiques)

```
> unDé.rouler()
Le dé roule...
...et s'arrête sur le 5
< undefined
> unDé.rouler()
Le dé roule...
...et s'arrête sur le 6
< undefined
> unSecondDé.rouler()
Le dé roule...
...et s'arrête sur le 20
< undefined
> |
```

TP 4 : polymorphisme

Exercice 1 : créer un objet voiture doté d'une méthode rouler()

Dans cet exercice, vous avez la possibilité de vérifier vos connaissances. Cherchez à faire par vous-même l'exercice, éventuellement en remontant dans l'historique ou dans les premières pages du cahier de tps avant de regarder la solution page suivante.

- > Créer un objet **voiture** avec un attribut **nom** initialisé à "**Choupette**"
 - ajouter une méthode **rouler()** affichant le message "**La voiture Choupette roule**"
 - penser à utiliser **this.nom** et non directement **Choupette** pour afficher ce message
 - essayer de réaliser seul cet exercice, la solution se trouve sur la page suivante

```
voiture = { nom: "Chouquette" }
voiture.rouler = function() {
    console.log("La voiture " + this.nom + " roule")
}
voiture.rouler()
```

```
> voiture = { nom: "Chouquette" }
< ▶ {nom: "Chouquette"}
> voiture.rouler = function() {
    console.log("La voiture " + this.nom + " roule")
}
< f () {
    console.log("La voiture " + this.nom + " roule")
}
> voiture.rouler()
    La voiture Chouquette roule
< undefined
> |
```

- noter que nous avons utilisé le même nom de méthode (**rouler**) que pour les dés
- nous avons maintenant 3 objets qui répondent tous les 3 au même message **rouler()**
- avec des comportements et donc des effets différents

Exercice 2 : illustrer la puissance du polymorphisme

- > Créer une fonction pour faire rouler un objet

```
function faireRouler(unObjet) {
    console.log("fonction faire rouler")
    unObjet.rouler()
}
```

- > Exécuter cette fonction avec **unDé** puis avec **voiture**

```
faireRouler(unDé)
faireRouler(voiture)
```

```
> faireRouler(unDé)
    fonction faire rouler
    Le dé roule...
    ...et s'arrête sur le 6
< undefined
> faireRouler(voiture)
    fonction faire rouler
    La voiture Chouquette roule
< undefined
> |
```

- la fonction traite indifféremment les deux objets
 - noter qu'elle pourra aussi traiter les futurs objets qui sauront comprendre le message **rouler()**
- > Créer un objet **ballon** avec une méthode **rouler()** affichant le message **"Le ballon roule"**

- noter qu'il est possible de créer l'objet directement avec sa méthode

```
ballon = {  
  rouler: function() {  
    console.log("Le ballon roule")  
  }  
}
```

```
> ballon = {  
  rouler: function() {  
    console.log("Le ballon roule")  
  }  
}  
< ▶ {rouler: f}
```

- > Faire rouler ce ballon

```
ballon.rouler()
```

```
> ballon.rouler()  
Le ballon roule  
< undefined
```

- > Vérifier que la fonction **faireRouler()** est bien capable de le prendre en compte

```
faireRouler(ballon)
```

```
> faireRouler(ballon)  
fonction faire rouler  
Le ballon roule
```

- noter que la fonction n'a pas été modifiée et prend en compte du code qui n'existait pas
- le polymorphisme est un outil puissant qui permet de construire du code évolutif

TP 5 : classes

Dans ce TP, nous quittons le monde de JavaScript pour passer au langage Java, un langage à base de classes. Noter que, dans le TP précédent, nous avons dû recopier la méthode **rouler()** dans **unSecondDé**. Il est très fréquent que de nombreux objets aient les mêmes méthodes. Les langages de classes sont conçus pour ce type de problème. Noter que les dernières versions de JavaScript ont évolué pour permettre de créer des classes.

Exercice 1 : installation de l'environnement Java

- > Vérifier ou installer une version du jdk (Java Development Kit)
 - ouvrir une **Invite de commande**
 - saisir la commande **javac -version**
 - si nécessaire, télécharger un **jdk** open source, par exemple <https://adoptopenjdk.net/>
 - préférer une version **zip** correspondant à la plateforme
 - dézipper à l'endroit souhaité (par exemple **C:\java**)
 - saisir le mot **variable** dans la zone de recherche de Windows 10
 - trouver **Modifier les variables d'environnement pour votre compte**
 - configurer la variable **JAVA_HOME** et le **Path**
 - **JAVA_HOME** doit désigner le répertoire où est installé Java
 - le **Path** doit contenir **%JAVA_HOME%\bin**
 - vérifier dans une **nouvelle Invite de commande** avec **javac -version**
- > Installer un IDE (environnement intégré) tel que **Eclipse**
 - préférer une version package sur <https://www.eclipse.org/downloads/packages>
 - le package **Eclipse IDE for Java Developers** est suffisant pour découvrir Java et Eclipse
 - dézipper par exemple dans **c:\eclipse**
 - exécuter le lanceur **eclipse.exe**
 - préférer un workspace sur **c:\workspace** (plus facile à retrouver plus tard)

Exercice 2 : recréer le système des dés en Java

Le langage Java est beaucoup moins permissif que JavaScript. C'est un langage à base de classes : pour créer des objets (instances), il faut d'abord définir des classes. Java est aussi moins dynamique que JavaScript. Le code doit être écrit dans des fichiers sources **.java** puis compilé dans des fichiers **.class** avant d'être exécuté par une machine virtuelle. Heureusement, les IDE comme Eclipse masquent une bonne partie de ces étapes.

- > Créer un projet **intro objet**
 - menu **File > New > Java Project** (ne pas créer de module-info)
- > Créer une classe **Dé**
 - sélectionner le projet **intro objet > clic-droit > New > Class**
 - dans le champ **Package >** saisir **com.oxiane.intro** (package = regroupement de classes)

- dans le champ **Name** > saisir **Dé**
- un fichier **Dé.java** est maintenant créé et un éditeur montre le code suivant :

```
package com.oxiane.intro;

public class Dé {
}
```

- compléter le code avec ses deux attributs **nbFaces** et **valeur**

```
package com.oxiane.intro;

public class Dé {
    int nbFaces = 6;
    int valeur;

    void rouler() {
        System.out.println("Le dé roule...");
        this.valeur = (int) (Math.random() * this.nbFaces) + 1;
        System.out.println("... et s'arrête sur le " + this.valeur);
    }
}
```

- noter que le code est assez similaire à celui de JavaScript mais Java introduit le **typage**
 - l'écriture s'appuie sur un objet **System.out** et sa méthode **println()**
 - **remarque** : ce code simplifié n'est pas représentatif des bonnes pratiques Java
- > Ajouter une méthode principale pour créer et tester deux dés à 6 faces
- insérer une ligne avant la dernière accolade et saisir **main** suivi de **Ctrl+Espace**
 - choisir la proposition **main – main method** avec **Entrée**
 - une méthode **main()** est alors ajoutée (définition de méthode principale, exécutable)
 - compléter cette méthode **main()** pour créer deux instances **unDé** et **unSecondDé**
 - ici encore, le code est assez proche de JavaScript avec de petites différences (typage, new...)

```
...
public static void main(String[] args) {
    Dé unDé = new Dé();
    Dé unSecondDé = new Dé();
    unDé.rouler();
    unSecondDé.rouler();
    System.out.println("Somme : "
                        + (unDé.valeur + unSecondDé.valeur));
}
...
```

- > Enregistrer le code
- icône disquette en haut à gauche ou plutôt **Ctrl-s**
 - noter qu'Eclipse compile alors le code (signale les erreurs de compilations)
- > Exécuter le code
- dans l'éditeur > **clic-droit** > **Run As** > **Java Application**

- la vue **Console** affiche les traces


```
Le dé roule...  
... et s'arrête sur le 4  
Le dé roule...  
... et s'arrête sur le 3  
Somme : 7
```

Exercice 2 : modélisation avec les classes

L'intérêt de la programmation avec les classes réside dans la capacité à modéliser le système. Le standard UML (Unified Modeling Language) est un langage graphique (et textuel dans sa version 2) qui permet de concevoir finement des systèmes orientés objet. Ce langage constitue avant tout une notation graphique pour représenter les éléments communs à tous les systèmes, orientés objets ou non.

Pour illustrer la modélisation, réfléchissons à un autre thème : le **jeu du nombre secret**. Dans ce jeu très simple, une personne pense à un **nombre secret**, par exemple entre **1** et **100**, qu'une autre personne s'efforce de deviner par propositions successives. Le premier n'accorde que 3 types de réponse : c'est **PLUS**, c'est **MOINS** ou c'est **TROUVE** et le jeu s'arrête alors. En général, on compte le nombre de tentatives. Réfléchir à une modélisation objet du jeu à l'aide de classes.

- > Recenser les « objets » ou plutôt « classes d'objets » constituant le jeu
 - le terme « objet » est ici à prendre au sens « concept réifié »
 - dans ce jeu, il y a deux personnes distinctes qui n'ont pas les mêmes caractéristiques
 - réfléchir aux responsabilités (rôles) de ces éléments est intéressant
 - une personne pense à un nombre, l'autre essaie de le deviner, les deux sont des joueurs
- > Trouver un nom pour distinguer ces deux éléments
 - on peut par exemple parler du « joueur qui pense » et du « joueur qui devine »
 - ou raccourcir en « penseur » et « devineur »
- > Représenter ces deux classes dans la notation UML
 - **UMLet** est un outil UML simple avec une version en ligne **UMLetino** (www.umletino.com)
 - double cliquer dans la palette pour poser une **SimpleClass**
 - dans la notation UML, on représente une classe par un rectangle avec le nom de la classe
 - modifier les caractéristiques de l'élément sélectionné avec le texte en bas à droite
 - renommer cette classe en **JoueurQuiPense**
 - double cliquer pour dupliquer cette classe et créer ainsi **JoueurQuiDevine**



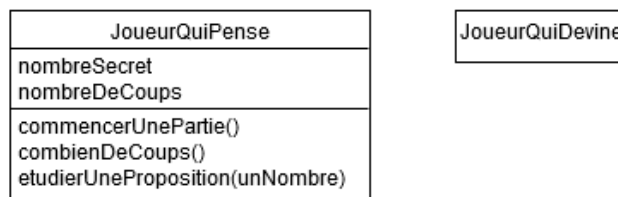
JoueurQuiPense

JoueurQuiDevine

- > La modélisation objet vise à se concentrer sur quelques caractéristiques importantes :
 - quels sont les noms et les responsabilités ?
 - quelles sont les propriétés (ou attributs) ?
 - quelles sont les opérations ?
 - quelles sont les relations et collaborations ?

> Commençons par le **JoueurQuiPense**

- il doit mémoriser un nombre secret qu'il déterminera aléatoirement au début de la partie
- il doit répondre aux propositions du **JoueurQuiDevine**
- il doit compter le nombre de tentatives
- pour réaliser tout ça, il lui faut deux propriétés : **nombreSecret** et **nombreDeCoups**
- une première méthode permettra de **commencerUnePartie()**
- à tout moment, il devrait pouvoir répondre à la question **combienDeCoups()**
- la troisième méthode consiste à **étudierUneProposition()** afin d'y répondre
- saisir ces informations dans UMLetino en séparant les trois compartiments par deux tirets --

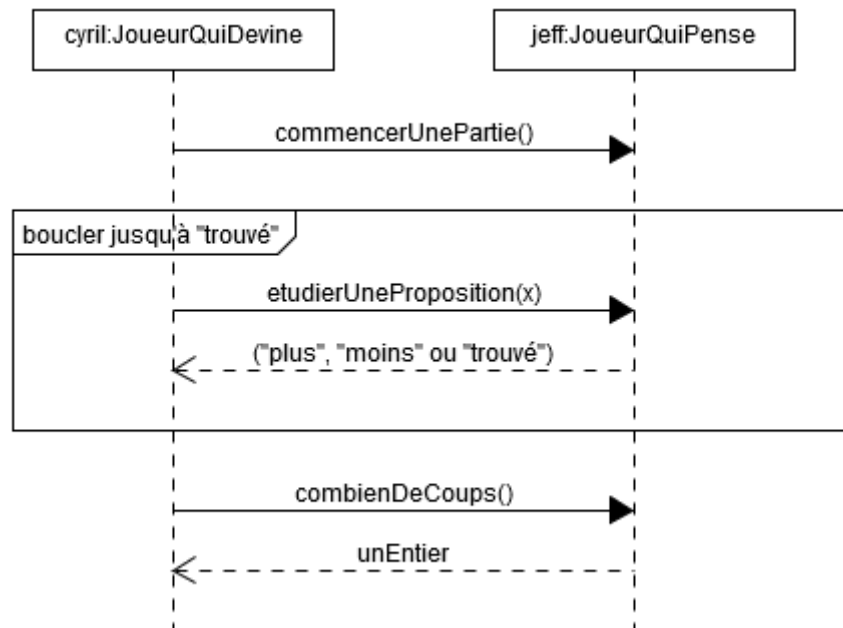


> Imaginer le scénario d'une partie

- le **JoueurQuiDevine** demande au **JoueurQuiPense** de **commencerUnePartie()**
- ce dernier détermine un **nombreSecret** au hasard et initialise le **nbDeCoups** à 0
- le **JoueurQuiDevine** demande au **JoueurQuiPense** d'**étudierUneProposition(x)**
- le **JoueurQuiPense** comptabilise l'essai en incrémentant le **nombreDeCoups**
- puis il compare **x** avec son **nombreSecret** afin de répondre (**PLUS, MOINS** ou **TROUVE**)
- le **JoueurQuiDevine** vérifie si la réponse est bien «**TROUVE** »
- dans le cas contraire, il recommence avec un autre nombre
- à la fin, il demande le **nombreDeCoups()** pour avoir son score

> (Optionnel) représenter ce scénario avec UML

- le diagramme de séquence d'UML représente les messages échangés entre des instances
- il représente les deux instances de joueurs, nommées ici **cyril** et **jeff**
- les flèches montrent les **messages** synchrones échangés et leur **réponse** en retour (si utile)
- le cadre montre une séquence en boucle



Exercice 3 : implémenter la classe JoueurQuiPense

- > Coder la classe **JoueurQuiPense** en Java
 - sur le projet **intro objet** dans l'arborescence > **clic-droit** > **New** > **Class**
 - saisir le **Name** avec **JoueurQuiPense**
 - définir les deux attributs et les méthodes
 - essayer tout seul avant de copier la solution page suivante
 - penser à écrire des traces dans la console pour faciliter la compréhension à l'exécution
 - **System.out.println();** (macro **sysout** + Ctrl-espace pour générer facilement cette instruction)
 - syntaxe Java de l'instruction conditionnelle : **if (condition) { instructions; }**

```
package com.oxiane.intro;

public class JoueurQuiPense {
    int nombreSecret;
    int nombreDeCoups;

    void commencerUnePartie() {
        System.out.println("Le joueur qui pense réfléchit à un nombre secret");
        nombreDeCoups = 0;
        nombreSecret = (int) (Math.random() * 100 + 1);
        System.out.println("Il pense à " + nombreSecret);
    }

    int combienDeCoups() {
        return nombreDeCoups;
    }

    String étudierUneProposition(int unNombre) {
        nombreDeCoups++;
        System.out.println("Le joueur qui pense étudie " + unNombre);
        if (unNombre > nombreSecret) {
            System.out.println("Il répond MOINS" );
            return "MOINS";
        }
        if (unNombre < nombreSecret) {
            System.out.println("Il répond PLUS" );
            return "PLUS";
        }
        System.out.println("Il répond TROUVE" );
        return "TROUVE";
    }
}
```

- **remarque** : ce code est simplifié et ne reflète pas les bonnes pratiques Java

- > Écrire une méthode **main** (méthode principale pour tester la classe)
 - insérer une ligne avant la dernière accolade
 - saisir **main** > **Ctrl-espace** > **main** – **main method** > **Entrée**
 - compléter le code pour créer une instance de **JoueurQuiPense** dans une variable **jqp**
 - envoyer les messages pour tester les 3 méthodes et écrire les résultats
 - essayer de le faire seul avant de regarder la solution ci-dessous

```
...
public static void main(String[] args) {
    JoueurQuiPense jqp = new JoueurQuiPense();
    jqp.commencerUnePartie();
    String réponse = jqp.étudierUneProposition(50);
    System.out.println(réponse);
    System.out.println(jqp.combinDeCoups());
}
}
```

- > Sauvegarder, exécuter (**clic-droit** > **Run As** > **Application Java**) et vérifier l'affichage

```
Le joueur qui pense réfléchit à un nombre secret
Il pense à 60
Le joueur qui pense étudie 50
Il répond PLUS
PLUS
1
```

Exercice 4 : réaliser le **JoueurQuiDevine**

> Analyser le comportement du **JoueurQuiDevine**

- avec le diagramme de séquence précédent, nous imaginons une première méthode générale
- le **JoueurQuiDevine** doit **dérouler une partie** contre un **JoueurQuiPense**
- nous pouvons donc définir une méthode **déroulerUnePartie()**
- on voit qu'il faudra désigner le **JoueurQuiPense** et donc passer une référence en paramètre

> Créer la classe **JoueurQuiDevine** et sa méthode **déroulerUnePartie()**

- sur le projet **intro objet** > clic-droit > **New** > **Class** > saisir **Name** avec **JoueurQuiDevine**

```
package com.oxiane.intro;

public class JoueurQuiDevine {

    void déroulerUnePartie() {
        System.out.println("Le joueur qui devine déroule une partie");
    }

}
```

> Prendre en compte le paramètre : il faut indiquer contre qui dérouler la partie

```
package com.oxiane.intro;

public class JoueurQuiDevine {

    void déroulerUnePartie(JoueurQuiPense adversaire) {
        System.out.println("Le joueur qui devine déroule une partie");
    }

}
```

> Traduire le diagramme de séquence précédent dans cette méthode

- on commencera par solliciter l'**adversaire** (le **JoueurQuiPense**) pour **commencerLaPartie()**
- il faudra ensuite le solliciter pour **étudierUneProposition()** et récupérer sa réponse
- et reboucler jusqu'à ce que la réponse soit **"TROUVE"**
- enfin, on lui demandera **combienDeCoups()** pour afficher le score
- commencer la traduction jusqu'à rencontrer un problème avec **etudierProposition()**
- essayer seul avant de regarder la solution page suivante

```
...
void déroulerUnePartie(JoueurQuiPense adversaire) {
    System.out.println("Le joueur qui devine déroule une partie");
    adversaire.commencerUnePartie();
    adversaire.étudierUneProposition(unNombre)
}
}
```

- problème : la méthode **étudierUneProposition()** requiert un **argument** (valeur de paramètre)
- il nous faut déterminer **unNombre** à proposer
- insérer une ligne pour déterminer ce nombre de manière aléatoire
- en profiter pour introduire des traces facilitant le suivi de l'exécution

```
...
void déroulerUnePartie(JoueurQuiPense adversaire) {
    System.out.println("Le joueur qui devine déroule une partie");
    adversaire.commencerUnePartie();

    System.out.println("Le joueur qui devine réfléchit à une proposition");
    int unNombre = (int) (Math.random() * 100 + 1);

    System.out.println("Il propose " + unNombre);
    adversaire.étudierUneProposition(unNombre);
}
}
```

> Récupérer la **réponse** du **JoueurQuiPense** et introduire la **boucle**

- pour des raisons techniques, la déclaration de la variable **réponse** doit se faire avant la boucle
- l'instruction Java pour répéter à la forme suivante : **do { instructions; } while (condition);**
- la comparaison entre chaîne de caractère se fait avec la méthode **equals()**
- la boucle doit recommencer jusqu'à ce que la **réponse** soit **différente** de **"TROUVE"**
- la négation en Java se note avec l'opérateur **!**
- on obtient donc le code suivant :

```
...
void déroulerUnePartie(JoueurQuiPense adversaire) {
    System.out.println("Le joueur qui devine déroule une partie");
    adversaire.commencerUnePartie();

    String réponse;
    do {
        System.out.println("Le joueur qui devine réfléchit à une proposition");
        int unNombre = (int) (Math.random() * 100 + 1);

        System.out.println("Il propose " + unNombre);
        réponse = adversaire.étudierUneProposition(unNombre);
    } while (!réponse.equals("TROUVE"));
}
}
```

> Il reste à gérer l'affichage du score

```
...
    } while(!réponse.equals("TROUVE"));
    int score = adversaire.combienDeCoups();
    System.out.println("En " + score + " coup(s)");
}
}
```

> Il nous faut maintenant écrire une méthode principale pour tester tout ça

- créer un **main** comme précédemment (saisir **main** avant la dernière accolade et **Ctrl-espace**)
- coder la création d'une instance de **JoueurQuiPense** et de **JoueurQuiDevine**
- et demander au **JoueurQuiDevine** de **déroulerUnePartie()**

```
...
public static void main(String[] args) {
    JoueurQuiPense jqp = new JoueurQuiPense();
    JoueurQuiDevine jqd = new JoueurQuiDevine();
    jqd.déroulerUnePartie(jqp);
}
}
```

> Sauvegarder (**Ctrl-s**) et exécuter (**clic-droit > Run As > Java Application**)

- observer le déroulement de la partie

```
Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 62
Le joueur qui devine réfléchit à une proposition
Il propose 1
Le joueur qui pense étudie 1
Il répond PLUS
Le joueur qui devine réfléchit à une proposition
Il propose 7
Le joueur qui pense étudie 7
Il répond PLUS
Le joueur qui devine réfléchit à une proposition
Il propose 99
Le joueur qui pense étudie 99
Il répond MOINS
Le joueur qui devine réfléchit à une proposition
Il propose 44
Le joueur qui pense étudie 44
Il répond PLUS
...
Le joueur qui devine réfléchit à une proposition
Il propose 47
Le joueur qui pense étudie 47
Il répond PLUS
Le joueur qui devine réfléchit à une proposition
Il propose 62
Le joueur qui pense étudie 62
Il répond TROUVE
En 69 coup(s)
```


- noter que le même nombre peut être proposé plusieurs fois
 - les réponses ne sont pas analysées, notre **JoueurQuiDevine** est complètement **aléatoire**
 - la programmation objet permet de structurer le code pour proposer plusieurs types de joueurs
- > Renommer la classe **JoueurQuiDevine** en **JoueurQuiDevineAléatoire**
- dans l'arborescence > sélectionner **JoueurQuiPense.java** > clic-droit > **Refactor** > **Rename**
 - saisir le nouveau nom **JoueurQuiDevineAléatoire**
 - noter que l'assistant renomme aussi les références dans le code
- > Réfléchir à différentes manières de jouer et trouver un nom pour la classe correspondante
- un **JoueurQuiDevineSystématique** énumérera tous les nombres en commençant par 1
 - un **JoueurQuiDevineMnésique** se souviendra des 5 derniers nombres proposés ou plus
 - un **JoueurQuiDevineHyperMnésique** se souviendra de tous les nombres qu'il a déjà proposés
 - un **JoueurQuiDevineFuté** analysera les réponses et réduira l'intervalle des propositions
 - un **JoueurQuiDevineDichotomique** fera de même et proposera le milieu de l'intervalle
 - il existe de nombreuses possibilités
- > Analyser le **JoueurQuiDevineFuté**
- le déroulement de la partie est sensiblement le même
 - noter cependant que ce type de joueur devra mémoriser les **bornes (inf et sup)** de l'intervalle
 - il doit aussi mémoriser la **proposition** qu'il a fait pour pouvoir **analyser la réponse** obtenue
 - il pourra alors modifier la borne correspondante pour réduire l'intervalle
- > Créer la classe **JoueurQuiDevineFuté**
- copier simplement **JoueurQuiDevineAléatoire.java** dans l'arborescence (clic-droit > **Copy**)
 - sélectionner **com.oxiane.intro** puis coller la copie (clic-droit > **Paste**)
 - modifier le nom pour **JoueurQuiDevineFuté**
 - **ATTENTION : la copie n'est pas sélectionnée automatiquement !!!**
- > Modifier le code de **JoueurQuiDevineFuté** pour retraduire l'analyse précédente
- ajouter les attributs **borneInf**, **borneSup** et **proposition**
 - **initialiser les bornes au début de la méthode déroulerUnePartie()**
 - la détermination de la proposition est maintenant un peu plus complexe
 - déporter ce calcul dans une nouvelle méthode **int déterminerProposition()**
 - de même, déporter l'analyse de la réponse dans une méthode **analyserRéponse()**
 - on peut ici choisir de recevoir la réponse en paramètre ou la promouvoir en attribut
 - essayer seul avant de regarder la solution page suivante
 - l'exécution en **mode pas à pas (Debug As > Java Application)** avec la touche **F6** peut aider
 - poser d'abord un point d'arrêt (dans la **marge gauche** > clic-droit > **Toggle Breakpoint**)

```
9  void déroulerUnePartie(JoueurQuiPense adversaire) {  
10     System.out.println("Le joueur qui devine déroule une partie");
```

```
package com.oxiane.intro;

public class JoueurQuiDevineFuté {

    int borneInf;
    int borneSup;
    int proposition;

    void déroulerUnePartie(JoueurQuiPense adversaire) {
        System.out.println("Le joueur qui devine déroule une partie");
        adversaire.commencerUnePartie();
        borneInf = 1;
        borneSup = 100;
        String réponse;
        do {
            System.out.println("Le joueur qui devine réfléchit à une proposition");
            proposition = déterminerProposition();

            System.out.println("Il propose " + proposition);
            réponse = adversaire.étudierUneProposition(proposition);
            System.out.println("Le joueur qui devine analyse la réponse");
            analyserRéponse(réponse);
        } while (!réponse.equals("TROUVE"));
        int score = adversaire.combienDeCoups();
        System.out.println("En " + score + " coup(s)");
    }

    int déterminerProposition() {
        return (int) (Math.random() * (borneSup - borneInf + 1)) + borneInf;
    }

    void analyserRéponse(String réponse) {
        if (réponse.equals("TROUVE")) {
            borneInf = borneSup = proposition;
        }
        if (réponse.equals("MOINS")) {
            borneSup = proposition - 1;
        }
        if (réponse.equals("PLUS")) {
            borneInf = proposition + 1;
        }
        System.out.println("Le nombre secret est entre " + borneInf + " et " + borneSup);
    }

    public static void main(String[] args) {
        JoueurQuiPense jqp = new JoueurQuiPense();
        JoueurQuiDevineFuté jqd = new JoueurQuiDevineFuté();
        jqd.déroulerUnePartie(jqp);
    }
}
```

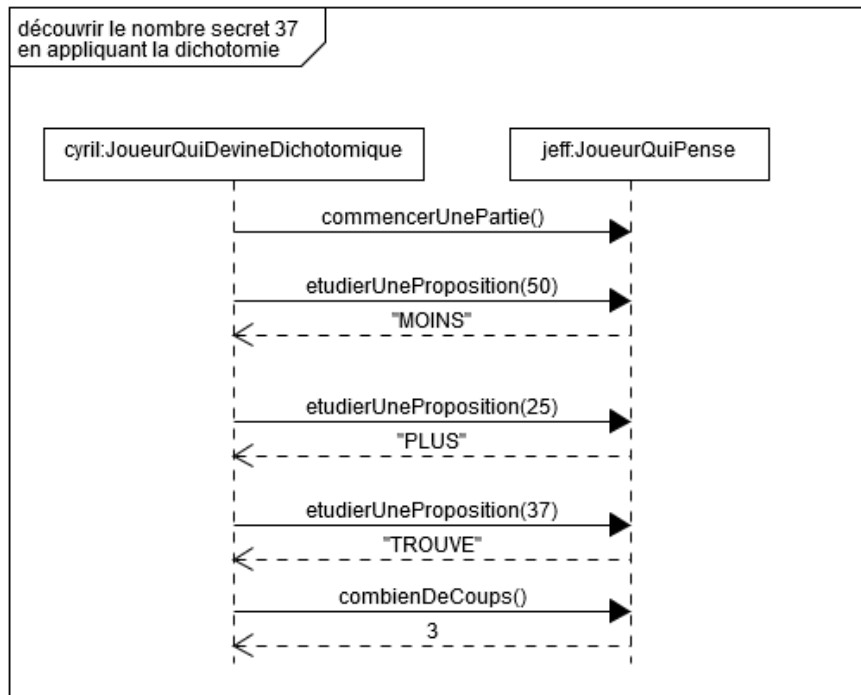
> Exécuter et vérifier le bon comportement

```
Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 65
Le joueur qui devine réfléchit à une proposition
Il propose 77
Le joueur qui pense étudie 77
Il répond MOINS
Le joueur qui devine analyse la réponse
Le nombre secret est entre 1 et 76
Le joueur qui devine réfléchit à une proposition
Il propose 30
Le joueur qui pense étudie 30
Il répond PLUS
...
Le joueur qui devine réfléchit à une proposition
Il propose 65
Le joueur qui pense étudie 65
Il répond TROUVE
Le joueur qui devine analyse la réponse
Le nombre secret est entre 65 et 65
En 6 coup(s)
```

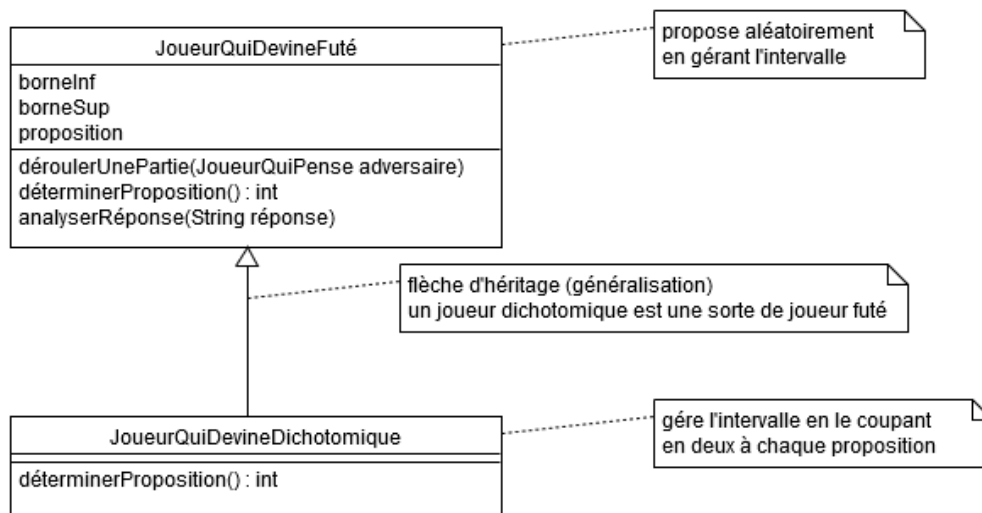
TP 6 : héritage

Exercice 1 : mettre en œuvre l'héritage

Notre classe **JoueurQuiDevineFuté** est manifestement plus efficace pour trouver le nombre secret grâce à la gestion de l'intervalle **borneInf** et **borneSup**. Remarquer qu'une autre manière de jouer s'appuie aussi sur cette gestion intelligente de l'intervalle, en mettant cette fois en œuvre la dichotomie. On coupe l'intervalle en deux à chaque essai au lieu de choisir au hasard.



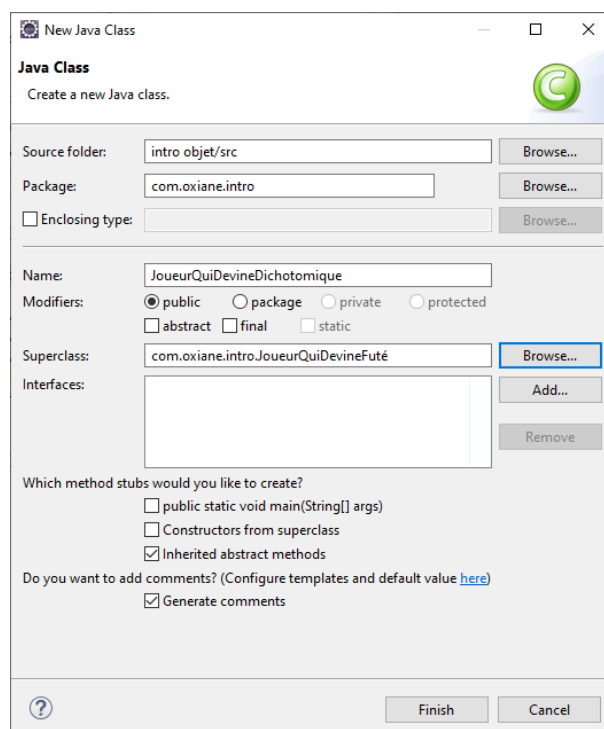
- > Observer à nouveau le code du **JoueurQuiDevineFuté**
 - remarquer que le code du **JoueurQuiDevineDichotomique** serait quasiment le même
 - la seule différence porte sur le calcul effectué dans la méthode **déterminerProposition()**
 - il serait tentant de copier la classe **JoueurQuiDevineFuté** et de l'adapter
 - mais la programmation objet offre un mécanisme beaucoup plus puissant : l'héritage
- > Observer comment UML nous permet de représenter la similitude entre les deux classes



- grâce à cette flèche spéciale (généralisation) on précise un héritage
- le joueur dichotomique hérite des attributs et des méthodes du joueur futé
- la seule différence : il redéfinit le comportement pour la méthode **déterminerProposition()**
- autrement dit, quand on lui demande **déroulerUnePartie()**, il fait comme pour le joueur futé
- mais si on lui demande **déterminerProposition()**, il fait à sa propre manière
- cette économie de conception (on conçoit en spécifiant le delta), on la retrouve dans le code

➤ Créer la classe **JoueurQuiDevineDichotomique** héritant (**extend**) de **JoueurQuiDevineFuté**

- sur le package com.oxiane.intro > clic-droit > New > Class
- saisir **Name** avec **JoueurQuiDevineDichotomique**
- saisir **Superclass** avec **JQDF > Ctrl-espace** (ou s'aider du bouton **Browse**)
- le champ doit alors contenir **com.oxiane.intro.JoueurQuiDevineFuté**



- observer la définition de la classe obtenue (la classe hérite de **JoueurQuiDevineFuté**)

```
package com.oxiane.intro;

public class JoueurQuiDevineDichotomique extends JoueurQuiDevineFuté
{
}
```

> Redéfinir la méthode **déterminerProposition()**

- pour cela, touche **Tab** > saisir **dét** > **Ctrl-espace** > choisir **déterminerProposition()**
- modifier le code généré pour définir le nouveau calcul (milieu de l'intervalle)
- noter que la division (**/ 2**) est une division entière en Java (ex : **1 / 2** donne 0)
- l'annotation **@Override** est un marqueur Java indiquant que cette méthode est une redéfinition

```
package com.oxiane.intro;

public class JoueurQuiDevineDichotomique extends JoueurQuiDevineFuté
{
    @Override
    int déterminerProposition() {
        return borneInf + (borneSup - borneInf) / 2;
    }
}
```

> Copier la méthode **main()** depuis la classe **JoueurQuiDevineFuté** et l'adapter

```
package com.oxiane.intro;

public class JoueurQuiDevineDichotomique extends JoueurQuiDevineFuté
{
    @Override
    int déterminerProposition() {
        return borneInf + (borneSup - borneInf) / 2;
    }

    public static void main(String[] args) {
        JoueurQuiPense jqp = new JoueurQuiPense();
        JoueurQuiDevineFuté jqd = new JoueurQuiDevineDichotomique();
        jqd.déroulerUnePartie(jqp);
    }
}
```

- noter qu'il n'est pas nécessaire de changer le type de la variable **jqd**
- un **JoueurQuiDevineDichotomique** est une sorte de **JoueurQuiDevineFuté** (un sous type)

> Sauvegarder et exécuter pour vérifier le comportement

```
Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 68
Le joueur qui devine réfléchit à une proposition
Il propose 50
Le joueur qui pense étudie 50
Il répond PLUS
```

```
Le joueur qui devine analyse la réponse
Le nombre secret est entre 51 et 100
Le joueur qui devine réfléchit à une proposition
Il propose 75
Le joueur qui pense étudie 75
Il répond MOINS
Le joueur qui devine analyse la réponse
Le nombre secret est entre 51 et 74
Le joueur qui devine réfléchit à une proposition
Il propose 62
Le joueur qui pense étudie 62
Il répond PLUS
Le joueur qui devine analyse la réponse
Le nombre secret est entre 63 et 74
Le joueur qui devine réfléchit à une proposition
Il propose 68
Le joueur qui pense étudie 68
Il répond TROUVE
Le joueur qui devine analyse la réponse
Le nombre secret est entre 68 et 68
En 4 coup(s)
```

- remarquer la faible quantité de code écrite (uniquement le delta) pour ce résultat

L'héritage est un mécanisme puissant de réutilisation. Bien utilisé, il peut faciliter la maintenance corrective et évolutive d'un système. Attention cependant, c'est une notion qui peut rapidement conduire à une grande complexité s'il est mal utilisé. Garder à l'esprit que l'héritage est avant tout une notion conceptuelle et non technique.

TP 7 : classes et méthodes abstraites

L'héritage est une notion puissante de conception qui peut être mise en œuvre de deux manières. Dans notre exemple précédent, nous l'avons mis en œuvre pour **spécialiser** le **JoueurQuiDevineFuté**. Nous sommes partis du **JoueurQuiDevineFuté** et avons imaginé une version légèrement différente, plus spécifique, le **JoueurQuiDevineDichotomique**. C'est une approche descendante : on part d'une classe et on dérive des sous-classes. Une autre approche possible, plus abstraite, consiste à observer des points communs entre des classes pour concevoir une **généralisation** des concepts. Cette approche amène souvent à la création de concept **abstrait**.

Exercice 1 : généraliser la notion de joueur qui devine

- > Observer nos 3 classes de joueur qui devine et réfléchir au **JoueurQuiDevineSystématique**
 - **JoueurQuiDevineAléatoire**, **JoueurQuiDevineFuté**, **JoueurQuiDevineDichotomique**
 - remarquer un schéma général qui se dégage
 - à chaque fois, le joueur qui devine déroule un traitement très similaire **déroulerUnePartie()**
 - **le déroulement se passe systématiquement en 3 phases :**
 - **déterminerUneProposition()**, appeler **étudierUneProposition()** puis **analyserRéponse()**
 - et on reboucle jusqu'à obtenir **"TROUVE"** avant de demander **combienDeCoups()**
 - nous pouvons construire une abstraction de **JoueurQuiDevine** générale
 - tout comme nous sommes habitués à parler de l'abstraction **Arbre**
 - c'est-à-dire l'ensemble général regroupant les chênes, hêtres, saules, peupliers, marronniers...
- > Créer la classe **JoueurQuiDevineAbstrait**
 - sur le package **com.oxiane.intro** > clic-droit > **New** > **Class**
 - saisir **Name** avec **JoueurQuiDevineAbstrait**
 - cocher la case **abstract**
- > Observer le code généré

```
package com.oxiane.intro;

public abstract class JoueurQuiDevineAbstrait {

}
```

 - Java utilise le mot **abstract** pour indiquer qu'une classe est abstraite
 - il interdira d'utiliser **new** pour créer des instances avec cette classe
 - une classe abstraite n'est là que pour définir un type abstrait et hériter de propriétés communes
- > Copier la méthode **déroulerUnePartie()** depuis la classe **JoueurQuiDevineFuté**
 - noter qu'Eclipse permet aussi de copier depuis l'arborescence
 - des erreurs apparaissent, le code est incomplet
- > Copier de la même manière l'attribut **proposition** depuis la classe **JoueurQuiDevineFuté**

- > Survoler l'erreur sur la ligne **proposition = déterminerProposition();**
 - cliquer sur la suggestion qui apparaît : **Create abstract method 'détermineProposition()'**
 - tous les joueurs qui devinent doivent déterminer une proposition mais chacun à sa manière
 - il n'y a aucune manière par défaut pour le faire, c'est pourquoi la méthode est **abstraite**
 - c'est-à-dire sans aucun corps

```
protected abstract int déterminerProposition();
```

- noter le modifieur **protected** ici, hors de propos et que l'on peut supprimer

- > Survoler l'erreur sur la ligne **analyserRéponse(réponse);**
 - cliquer sur la suggestion **Create method 'analyserRéponse(String)'**
 - cette fois, la méthode est générée avec un corps vide et un commentaire

```
private void analyserRéponse(String réponse) {  
    // TODO Auto-generated method stub  
}
```

- en effet, le comportement par défaut consistant à ne rien faire pour est acceptable
- supprimer le modifieur **private**, hors de propos pour ce cours et modifier le commentaire

```
void analyserRéponse(String réponse) {  
    // ne rien faire par défaut  
}
```

- > Corriger les deux dernières erreurs sur les lignes **borneInf = 1; et borneSup = 100;**
 - noter que certains joueurs ont besoin de se préparer avant de jouer et d'autres non
 - nous pouvons introduire une phase supplémentaire optionnelle pour se préparer
 - la méthode peut reprendre le nom **commencerUnePartie()** (polymorphisme)
 - remplacer les deux lignes par : **this.commencerUnePartie();**
 - noter que **this** est optionnel mais permet ici de bien distinguer les deux appels
 - survoler l'erreur et cliquer sur la suggestion **Create method 'commencerUnePartie()'**
 - corriger pour enlever le private et remplacer le commentaire par une trace

```
void commencerUnePartie() {  
    System.out.println("Le joueur qui devine se prépare");  
}
```

- le code complet devrait ressembler à :

```
package com.oxiane.intro;  
  
public abstract class JoueurQuiDevineAbstrait {  
  
    int proposition;  
  
    void déroulerUnePartie(JoueurQuiPense adversaire) {  
        System.out.println("Le joueur qui devine déroule une  
partie");  
        adversaire.commencerUnePartie();  
    }  
}
```

```
this.commencerUnePartie();
String réponse;
do {
    System.out.println("Le joueur qui devine réfléchit à une proposition");
    // int unNombre = (int) (Math.random() * 100 + 1);
    proposition = déterminerProposition();

    System.out.println("Il propose " + proposition);
    réponse = adversaire.étudierUneProposition(proposition);
    System.out.println("Le joueur qui devine analyse la réponse");
    analyserRéponse(réponse);
} while (!réponse.equals("TROUVE"));
int score = adversaire.combienDeCoups();
System.out.println("En " + score + " coup(s)");
}

void commencerUnePartie() {
    System.out.println("Le joueur qui devine se prépare");
}

abstract int déterminerProposition();

void analyserRéponse(String réponse) {
    // ne rien faire par défaut
}
}
```

Exercice 2 : utiliser la classe abstraite pour créer le JoueurQuiDevineSystématique

- > Déplier le fichier **JoueurQuiDevineAbstrait.java**
 - observer le **A** sur l'icône de la classe indiquant qu'il s'agit d'une classe abstraite
- > Créer une sous-classe **JoueurQuiDevineSystématique**
 - sur la classe **JoueurQuiDevineAbstrait** > clic-droit > New > Class
 - noter **Superclass** est déjà renseigné sur **com.oxiane.intro.JoueurQuiDevineAbstrait**
 - saisi **Nom** avec **JoueurQuiDevineSystématique**
 - vérifier que la case à cocher **Inherited abstract methods** est bien cochée
- > Observer le code généré

```
package com.oxiane.intro;

public class JoueurQuiDevineSystématique extends
JoueurQuiDevineAbstrait {

    @Override
    int déterminerProposition() {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

```
}
```

- la classe hérite de **JoueurQuiDevineAbstrait**
- la méthode **déterminerProposition()** qui était abstraite est redéfinie car elle est obligatoire
- les autres méthodes sont héritées avec leur comportement par défaut

> Compléter la classe pour définir le comportement du **JoueurQuiDevineSystématique**

- il mémorise sa **prochaineProposition** en commençant à 1
- c'est cette **prochaineProposition** qui est retournée lors du **déterminerProposition()**
- à chaque réponse "**PLUS**", il **incrémente** cette **prochaineProposition**

```
package com.oxiane.intro;

public class JoueurQuiDevineSystématique extends
    JoueurQuiDevineAbstrait {

    int prochaineProposition;

    @Override
    void commencerUnePartie() {
        super.commencerUnePartie();
        prochaineProposition = 1;
    }

    @Override
    int déterminerProposition() {
        return prochaineProposition;
    }

    @Override
    void analyserRéponse(String réponse) {
        if (réponse.equals("PLUS")) {
            prochaineProposition++;
        }
    }
}
```

- noter l'utilisation du **super.commencerUnePartie()**; pour invoquer la méthode héritée
- la méthode est ainsi redéfinie partiellement, en la complétant

> Recopier et modifier la méthode **main()** depuis la classe **JoueurQuiDevineFuté**

```
public static void main(String[] args) {
    JoueurQuiPense jqp = new JoueurQuiPense();
    JoueurQuiDevineAbstrait jqd =
        new JoueurQuiDevineSystématique();
    jqd.déroulerUnePartie(jqp);
}
```

- ici encore, la variable **jqd** peut être de type **JoueurQuiDevineAbstrait**, le type le plus général

> Sauvegarder et exécuter pour observer le bon déroulement

```
Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 17
Le joueur qui devine se prépare
Le joueur qui devine réfléchit à une proposition
Il propose 1
Le joueur qui pense étudie 1
Il répond PLUS
Le joueur qui devine analyse la réponse
Le joueur qui devine réfléchit à une proposition
Il propose 2
Le joueur qui pense étudie 2
Il répond PLUS
Le joueur qui devine analyse la réponse
...
Le joueur qui devine réfléchit à une proposition
Il propose 17
Le joueur qui pense étudie 17
Il répond TROUVE
Le joueur qui devine analyse la réponse
En 17 coup(s)
```

- noter la ligne **Le joueur qui devine se prépare**, présente grâce à l'appel au **super**

Exercice 3 : réorganiser le code (refactoring)

- > Réorganiser la classe **JoueurQuiDevineFuté** pour la faire hériter de **JoueurQuiDevineAbstrait**
 - il faudra ajouter le **extends JoueurQuiDevineAbstrait**
 - supprimer la propriété **proposition** et la méthode **déroulerUnePartie()** maintenant héritées
 - redéfinir la méthode **commencerUnePartie()** pour initialiser les bornes
 - ajouter l'annotation **@Override** pour indiquer les méthodes maintenant redéfinies

```
package com.oxiane.intro;

public class JoueurQuiDevineFuté extends JoueurQuiDevineAbstrait {

    int borneInf;
    int borneSup;

    @Override
    void commencerUnePartie() {
        super.commencerUnePartie();
        borneInf = 1;
        borneSup = 100;
    }

    @Override
    int déterminerProposition() {
        return (int) (Math.random() * (borneSup - borneInf + 1)) +
        borneInf;
    }

    @Override
```

```

void analyserRéponse(String réponse) {
    if (réponse.equals("TROUVE")) {
        borneInf = borneSup = proposition;
    }
    if (réponse.equals("MOINS")) {
        borneSup = proposition - 1;
    }
    if (réponse.equals("PLUS")) {
        borneInf = proposition + 1;
    }
    System.out.println("Le nombre secret est entre " + borneInf +
" et " + borneSup);
}

public static void main(String[] args) {
    JoueurQuiPense jqp = new JoueurQuiPense();
    JoueurQuiDevineFuté jqd = new JoueurQuiDevineFuté();
    jqd.déroulerUnePartie(jqp);
}
}

```

- > Sauvegarder et exécuter pour vérifier que la classe continue de fonctionner
 - noter là encore l'apparition de la ligne **Le joueur qui devine se prépare**

```

Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 80
Le joueur qui devine se prépare
Le joueur qui devine réfléchit à une proposition
Il propose 24
Le joueur qui pense étudie 24
Il répond PLUS
Le joueur qui devine analyse la réponse
Le nombre secret est entre 25 et 100
...
Le nombre secret est entre 80 et 80
En 7 coup(s)

```

- > Modifier la classe **JoueurQuiDevineAléatoire** pour hériter de **JoueurQuiDevineAbstrait**

```

public class JoueurQuiDevineAléatoire extends
JoueurQuiDevineAbstrait {

```

- noter l'erreur indiquant que la classe doit implémenter la méthode **déterminerProposition()**
- cliquer sur la suggestion **Add unimplemented methods**
- compléter la méthode en reprenant le code actuel pour déterminer un nombre aléatoire
- supprimer la méthode **déroulerUnePartie()** maintenant héritée
- le code final devient :

```

public class JoueurQuiDevineAléatoire extends
JoueurQuiDevineAbstrait {

    @Override

```

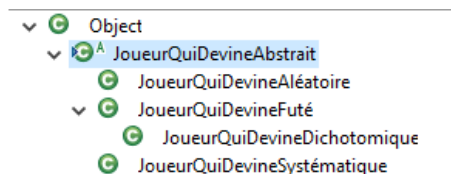
```
int déterminerProposition() {
    return (int) (Math.random() * 100 + 1);
}

public static void main(String[] args) {
    JoueurQuiPense jqp = new JoueurQuiPense();
    JoueurQuiDevineAléatoire jqd = new JoueurQuiDevineAléatoire();
    jqd.déroulerUnePartie(jqp);
}
}
```

> Sauvegarder et exécuter pour vérifier le bon fonctionnement

```
Le joueur qui devine déroule une partie
Le joueur qui pense réfléchit à un nombre secret
Il pense à 70
Le joueur qui devine se prépare
Le joueur qui devine réfléchit à une proposition
Il propose 8
Le joueur qui pense étudie 8
Il répond PLUS
Le joueur qui devine analyse la réponse
...
Le joueur qui devine réfléchit à une proposition
Il propose 70
Le joueur qui pense étudie 70
Il répond TROUVE
Le joueur qui devine analyse la réponse
En 26 coup(s)
```

- > Observer la hiérarchie de classe ayant pour racine JoueurQuiDevineAbstrait
- sélectionner **JoueurQuiDevineAbstrait** > clic-droit > **Open Type Hierarchy**



Exercice 4 : créer une nouvelle classe de joueur, le JoueurQuiDevineInteractif

Le principe est d'offrir la possibilité à un joueur humain de deviner le nombre secret. La classe servira de relai entre le programme et l'utilisateur. L'objectif ici est d'apercevoir les possibilités de réutilisation offertes par la programmation orientée objet. Le développement d'applications réelles se passe de la même manière que nos exemples précédents. Des classes existantes sont proposées pour offrir toutes sortes de services et le développeur les réutilise, par composition ou par héritage, afin de construire son application.

- > Créer la classe **JoueurQuiDevineInteractif** héritant de **JoueurQuiDevineAbstrait**
- sélectionner la classe **JoueurQuiDevineAbstrait** > clic-droit > **New > Class**
 - saisir le **Name** avec **JoueurQuiDevineInteractif**
 - compléter les opérations **déterminerProposition()** et **analyserRéponse()**

- nous utilisons une classe standard Java qui permet de construire des boîtes de dialogue
- recopier et adapter un main() pour permettre l'exécution d'un test

```
package com.oxiane.intro;

import javax.swing.JOptionPane;

public class JoueurQuiDevineInteractif extends
JoueurQuiDevineAbstrait {

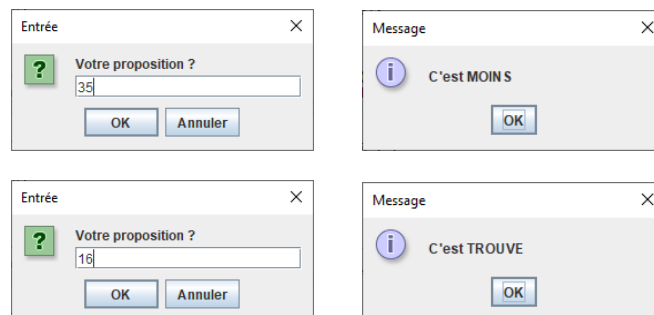
    @Override
    int déterminerProposition() {
        String uneString = JOptionPane.showInputDialog(
            "Votre proposition ?");
        return Integer.parseInt(uneString);
    }

    @Override
    void analyserRéponse(String réponse) {
        JOptionPane.showMessageDialog(null, "C'est " + réponse);
    }

    public static void main(String[] args) {
        JoueurQuiPense jqp = new JoueurQuiPense();
        JoueurQuiDevineAbstrait jqd = new JoueurQuiDevineInteractif();
        jqd.déroulerUnePartie(jqp);
    }
}
```

- noter l'utilisation des méthodes de la classe **JOptionPane** et **Integer**

> Sauvegarder et exécuter pour vérifier le bon fonctionnement



TP 8 : expérimenter les concepts avancés

Dans ce TP optionnel, nous expérimentons des concepts plus avancés comme la notion d'interface et de collection.

- > Créer une classe **JoueurQuiDevineMnésique** héritant de **JoueurQuiDevineAléatoire**
 - ici, l'idée est de se souvenir des propositions passées pour éviter de les proposer
 - il nous faut donc mémoriser la liste des **propositionsPassées**
 - et éviter de proposer un nombre déjà proposé lorsqu'on détermine une proposition
 - le concept de liste d'objets est suffisamment général pour faire l'objet de classes existantes
 - il peut être intéressant de concevoir soi-même une telle classe
 - de nombreuses possibilités existent pour implémenter une solution
 - c'est pourquoi la notion d'interface permet de définir un **type complètement abstrait**
 - typiquement, on utilisera en Java la notion de **Collection** (une **interface**)
 - et selon, la situation, une des nombreuses implémentations **ArrayList**, **HashSet**, **LinkedList**...

```
package com.oxiane.intro;

import java.util.ArrayList;
import java.util.Collection;

public class JoueurQuiDevineMnésique extends
JoueurQuiDevineAléatoire {

    Collection<Integer> propositionsPassées;

    @Override
    void commencerUnePartie() {
        super.commencerUnePartie();
        propositionsPassées = new ArrayList<Integer>();
    }

    @Override
    int déterminerProposition() {
        int p;
        do {
            p = super.déterminerProposition();
        } while (déjàProposé(p));
        propositionsPassées.add(p);
        return p;
    }

    boolean déjàProposé(int unNombre) {
        return propositionsPassées.contains(unNombre);
    }

    public static void main(String[] args) {
        JoueurQuiPense jqp = new JoueurQuiPense();
        JoueurQuiDevineAbstrait jqd = new JoueurQuiDevineMnésique();
    }
}
```



```
jqd.déroulerUnePartie(jqp);  
}  
}
```

- noter les **imports** permettant d'indiquer qu'on réutilise des classes d'autres packages
 - l'utilisation de l'interface **Collection<Integer>** (interface générique pour contenir des entiers)
 - l'implémentation choisie (**ArrayList**) dans la méthode **commencerUnePartie()**
 - la redéfinition de la méthode **déterminerProposition()** pour éviter de reproposer un nombre
 - l'appel à **super** pour faire comme au dessus (prendre un nombre au hasard entre 1 et 100)
 - l'ajout d'une méthode **déjàProposé()** pour vérifier si un nombre a déjà été proposé
 - son code s'appuyant sur une méthode déjà existante de l'interface **Collection** (**contains()**)
- > Sauvegarder et exécuter pour vérifier le bon fonctionnement
- il ne devrait pas y avoir de doublon (difficile à vérifier)

> Changer d'implémentation pour utiliser un **HashSet** au lieu d'une **ArrayList**

```
package com.oxiane.intro;  
  
import java.util.Collection;  
//import java.util.ArrayList;  
import java.util.HashSet;  
  
public class JoueurQuiDevineMnésique extends  
JoueurQuiDevineAléatoire {  
  
    Collection<Integer> propositionsPassées;  
  
    @Override  
    void commencerUnePartie() {  
        super.commencerUnePartie();  
        //propositionsPassées = new ArrayList<Integer>();  
        propositionsPassées = new HashSet<Integer>();  
    }  
    ...  
}
```

- > Sauvegarder et vérifier le comportement identique
- la classe **HashSet** est censée être un peu plus rapide (mais c'est difficile à voir ici)