

Contents

1	Les objets	3
1.1	Introduction	3
1.2	Rappel sur les objets	3
1.2.1	Classes et instances	3
1.2.2	La classe modèle de l'objet	3
1.2.3	Créer des instances	4
1.2.4	Champs statiques	5
1.2.5	Publique et privé	6
1.3	Le modèle objet : diviser pour régner	7
1.3.1	Un exemple : Puissance 4	7
1.3.2	Le tableau	7
1.3.3	Le jeton	7
1.3.4	Le joueur humain	7
1.3.5	Le joueur informatique	8
1.4	Création du squelette des classes	8
1.4.1	Création de la classe Tableau dans Eclipse	8
1.4.2	Le tableau	9
1.4.3	Le jeton	10
1.4.4	Le joueur humain	10
1.4.5	Le joueur informatique	10
1.4.6	Et maintenant ?	10
1.5	Contenu des méthodes	11
1.5.1	Cas d'utilisation normal	11
1.6	Conclusion	12
1.7	Exercices	12

Cours 1

Les objets

La version électronique de ce cours est disponible à l'URL http://www.derepas.com/java_int.

La version d'Eclipse utilisée pour le cours est la 3.3.1.1.

1.1 Introduction

Le but cours de Java intermédiaire est double :

- savoir découper un cas pratique sous forme de classes. Ainsi à partir d'exigences fonctionnelles décrivant une application à réaliser vous serez capable de créer des prototypes de classes correspondant à ce découpage.
- une fois que l'on dispose de la structure des classes il faut ensuite savoir implémenter les traitements à réaliser au sein de chaque classe.

1.2 Rappel sur les objets

Cette section contient des rappels sur les objets en Java. Java est un langage orienté objet. En Java tout est pris dans ce que l'on appelle des objets. Aussi est-il important de savoir décomposer un problème en une série d'objets permettant de le résoudre.

1.2.1 Classes et instances

Les objets (au sens java) sont mis en oeuvre à l'aide des notions suivantes :

- la classe qui définit une sorte de modèle, un peu comme le plan d'un architecte.
- les instances représentent des réalisations conformément au plan.

1.2.2 La classe modèle de l'objet

Ainsi la classe est une sorte de modèle de l'objet. Pour cela elle va définir :

- les données contenues, dans les champs de la classe,
- les moyens de changer ces données, dans les méthodes de la classe.

Une méthode spéciale permettant d'initialiser l'objet est nommée constructeur et ne retourne pas d'argument.

Voici un exemple : nous allons définir une classe contenant un entier, un constructeur, une méthode pour récupérer la valeur de cet entier, et une dernière pour fixer une nouvelle valeur.

```
1  /**
2   * Définition d'une nouvelle classe.
3   */
4  class ASimpleClass {
5      /**
6       * Champ i contenant la donnée.
```

```

7     */
8     int i;
9     /**
10    * Constructeur fixant la valeur initiale de i
11    */
12    ASimpleClass (int j) {
13        i=j;
14    }
15    /**
16    * Méthode renvoyant la valeur de i
17    */
18    int getValue() {
19        return i;
20    }
21    /**
22    * Méthode assignant une nouvelle valeur à i
23    */
24    void setValue(int j) {
25        i=j;
26    }
27 }

```

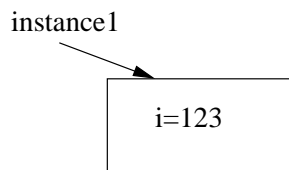
1.2.3 Créer des instances

Le plan que nous venons d'établir est bien joli, mais pour le moment rien n'existe : c'est juste un plan. Si on souhaite le réaliser il faut créer de nouvelles instances à l'aide du mot clé `new`.

Voici comment créer de nouvelles instances de la classe `ASimpleClass` :

```
ASimpleClass instance1 = new ASimpleClass(123);
```

Que s'est-il passé dans la machine ? Un espace mémoire a été réservé pour conserver l'information relative à l'instance `instance1`.

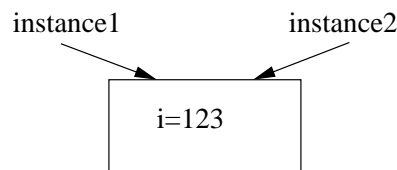


De l'espace mémoire est réservé pour `instance1`.

Si on ne redemande pas de l'espace à la machine avec `new` on ne peut conserver qu'une seule instance. Ainsi on peut renommer l'instance précédente :

```
ASimpleClass instance2 = instance1;
```

Voici le schéma mémoire associé `instance1` et `instance2` sont en fait la même chose :

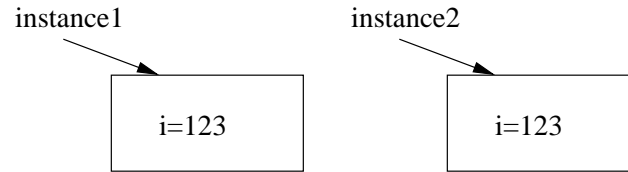


`instance1` et `instance2` sont en fait la même chose.

Puis on peut maintenant donner une nouvelle valeur à `instance2` :

```
instance2 = new Instance(123);
```

Voici le schéma mémoire associé :



instance1 et instance2 sont différents.

On peut maintenant modifier instance1 sans que instance2 ne soit impacté.

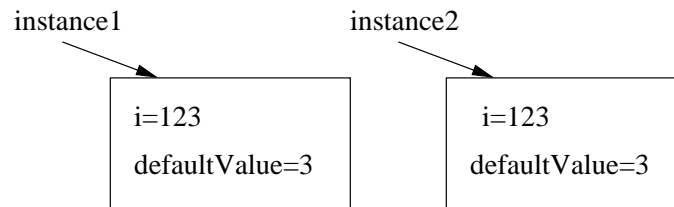
1.2.4 Champs statiques

Parfois on a envie qu'une donnée soit partagée entre toutes les instances d'une même classe, par exemple la valeur d'initialisation `initValue` dans la classe `ASimpleClass2`:

```

1 class ASimpleClass2 {
2     int i;
3     int defaultValue = 3;
4     ASimpleClass2() {
5         i=defaultValue;
6     }
7     int getValue() {
8         return i;
9     }
10 }
```

Le problème c'est que pour chaque instance on va stocker deux entiers : `i` et `defaultValue`. C'est illustré par le schéma ci-dessous :



Le champ `defaultValue` est présent dans chaque instance.

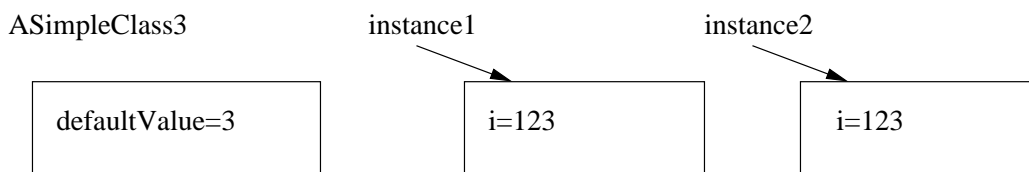
Or si on a beaucoup d'instances on va utiliser inutilement beaucoup de mémoire en mémorisant à chaque fois la valeur de `initValue` qui est en fait commune à toutes les instances.

Pour "factoriser" le champ `defaultValue` il faut rajouter l'attribut `static` devant le type, comme ci-dessous :

```

1 class ASimpleClass3 {
2     int i;
3     static int defaultValue = 3;
4     ASimpleClass3() {
5         i=defaultValue;
6     }
7     int getValue() {
8         return i;
9     }
10 }
```

La représentation mémoire est décrite ci-dessous :



Le champ `static defaultValue` est "mutualisé" entre les instances.

Exemple: Quand on affiche une chaîne de caractères à l'aide de la commande `System.out.println("Hello World!")` on appelle la méthode `println` sur l'objet `out` qui est un champ statique de la classe `System`. Il n'y a pas besoin de manipuler d'instance de la classe `System`.

Si on sait que la valeur d'initialisation `defaultValue` sera inchangée pendant l'exécution du programme on peut de plus rajouter l'attribut `final` qui interdit sa modification :

```
final static int defaultValue = 3;
```

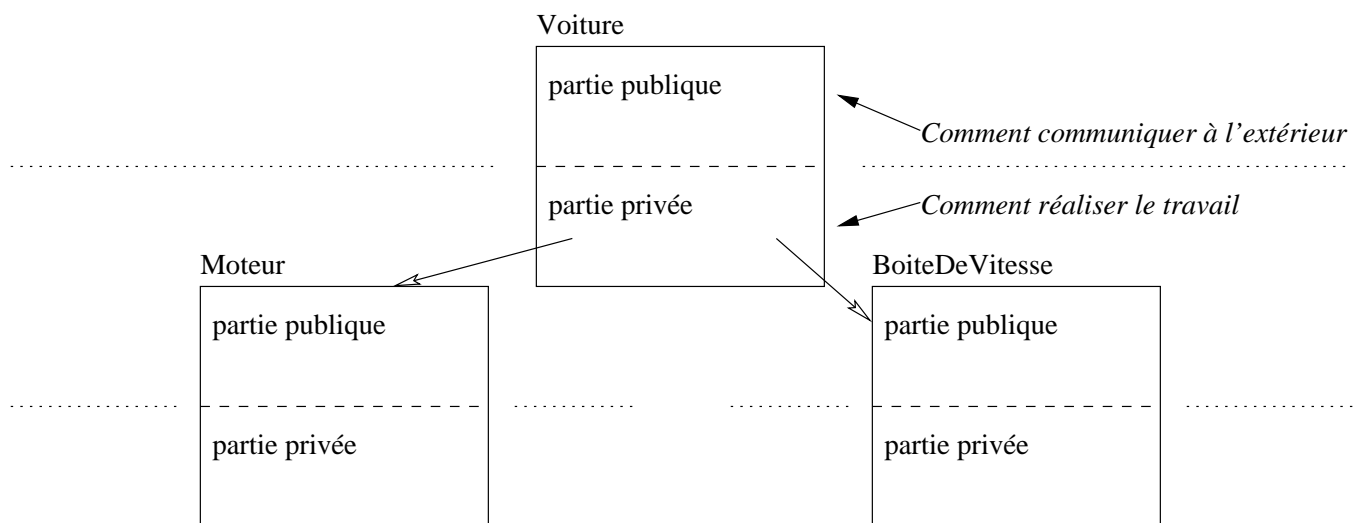
Dès lors toute modification de `defaultValue` provoquera une erreur à la compilation.

1.2.5 Publique et privé

Une notion importante est celle de champs ou méthodes publiques ou privés. Une méthode ou un champ public peut être vu ou modifié par les méthodes des classes extérieures à la classe courante. En revanche les variables privées ne peuvent être modifiées que par des méthodes de la classe courante.

Considérons une classe `Voiture` modélisant une voiture qui peut démarrer et s'arrêter qui comprend une boîte de vitesse et un moteur. Vu de l'extérieur on ne souhaite pas nécessairement exposer le fait que la voiture comprend une boîte de vitesse et un moteur, peut-être parce qu'elle pourrait contenir deux moteurs dans une nouvelle version, ou que plus tard on pense changer le type de boîte de vitesse. Aussi peut-on écrire la structure suivante

```
1 class Voiture {
2     BoiteDeVitesse maBoite; // champ privé
3     Moteur monMoteur; // champ privé
4     public Voiture() { // méthode publique
5         // ...
6     }
7     public void start() { // méthode publique
8         monMoteur.start();
9     }
10    public void stop() { // méthode publique
11        monMoteur.stop();
12    }
13 }
```



Notion de variable publique et privée.

Il faut bien comprendre que cette notion de variable publique et de variable privée est centrale dans le modèle objet. Elle permet de "passer à l'échelle". En effet on peut ainsi construire progressivement une hiérarchie d'objets en spécifiant à l'aide des parties publiques comment ces derniers vont être manipulés. Puis ensuite détailler comment ils vont réaliser leur tâches en écrivant les méthodes privées, ce qui nous amènera peut-être alors à créer de nouvelles classes.

1.3 Le modèle objet : diviser pour régner

Le modèle objet nous permet de découper un problème en sous problèmes. Nous proposons ici une illustration à travers un exemple.

1.3.1 Un exemple : Puissance 4

Nous nous proposons de faire un jeu de puissance 4, d'un humain contre la machine. Nous allons détailler la hiérarchie des objets à mettre en oeuvre pour réaliser un tel programme.

L'idée est de décomposer le jeu en ensembles ayant des responsabilités bien définies. L'idée est de procéder "en gros", on se réserve le droit de rajouter de nouveaux objets ou de détailler plus tard chaque objet. On peut par exemple choisir comme ensembles :

- le tableau contenant les jetons,
- le joueur humain,
- le joueur informatique.

L'important est maintenant de clairement déterminer pour chaque ensemble :

- son rôle,
- les données conservées,
- quelles sont les interactions avec les autres.

1.3.2 Le tableau

Son rôle est de représenter le tableau physique pour savoir où sont les jetons.

Les données conservées sont donc l'ensemble des cases avec la position des jetons.

Les interactions avec les autres sont :

- on veut pouvoir afficher le tableau à l'écran.
- on veut pouvoir rajouter un jeton dans une colonne.
- on veut pouvoir interroger le tableau pour savoir si dans une case donnée il y a un jeton et si oui de quelle couleur.
- on veut savoir si une colonne est pleine
- on veut savoir si le jeu est terminé.
- on veut pouvoir afficher le tableau.

On remarque de l'on parle d'un jeton. Il serait donc bon de formaliser cette notion avec une classe.

1.3.3 Le jeton

Son rôle est de représenter un jeton physique.

La donnée conservée est simplement la couleur du jeton : rouge ou jaune.

Les interactions avec les autres sont :

- créer un jeton d'une couleur donnée.
- demander la couleur d'un jeton.

1.3.4 Le joueur humain

Son rôle est de pouvoir donner à un humain la possibilité de jouer.

Il n'y a a priori pas de données conservées.

Les interactions avec les autres sont :

- demander à l'humain de jouer.

Il faut donc en fait quand même conserver une donnée : une référence au tableau qui stocke les jetons. Ce sera l'unique champ de la classe.

1.3.5 Le joueur informatique

Son rôle est de pouvoir permettre à un programme de jouer.

Comme dans le cas du joueur humain il faudra conserver une unique donnée : une référence au tableau qui stocke les jetons.
Les interactions avec les autres sont :

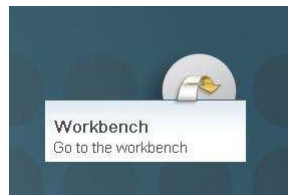
- demander où l'ordinateur de jouer.

1.4 Création du squelette des classes

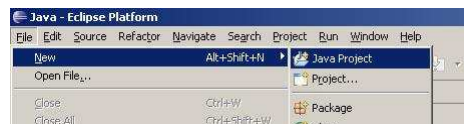
Nous avons maintenant une vue claire de la façon dont les classes vont s'agencer et quels sont leurs rôles respectifs. Nous allons pouvoir créer le squelette des classes.

1.4.1 Création de la classe Tableau dans Eclipse

Voici les étapes à réaliser dans l'interface NetBeans 3.2.2 pour créer la classe Tableau :



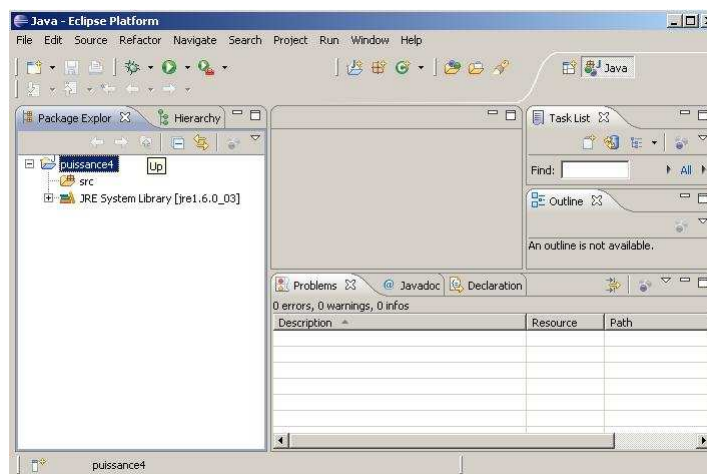
Si besoin afficher le "workbench".



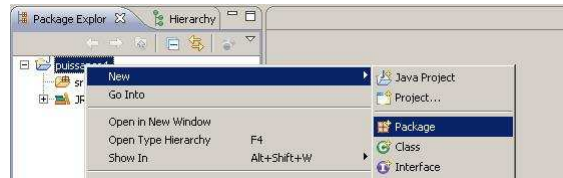
Dans le menu "file" choisir "New" puis "Java Project"



Rentrer le nom du projet et cliquer sur terminer.



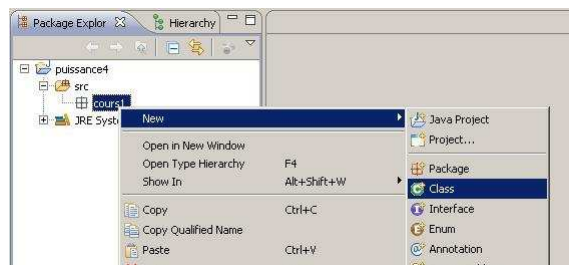
On obtient alors un écran comme ci-dessus



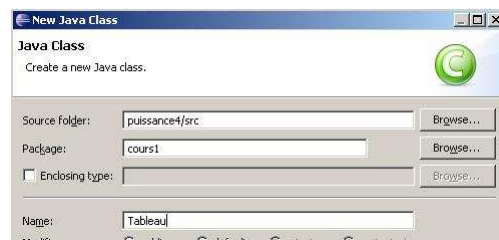
Cliquer droit sur le package choisir “New” et “Package”



Rentrer un nom pour le package, par exemple “cours1”



Cliquer droit sur le projet choisir “New” et “Class”



Rentrer ici le nom de la classe “Tableau”

1.4.2 Le tableau

Conformément aux données de la section précédente nous pouvons déclarer le prototype :

```

1  class Tableau {
2      Jeton donnees[][];
3      public Tableau() {
4      }
5      public void ajouteJeton (Jeton jeton, int colonne) {
6      }
7      public boolean jeuTermine() {
8      }
9      public void affiche() {
10     }
11     public Jeton quelJeton(int colonne, int rangee) {
12     }
13 }

```

On constate que si l’on fait un tableaux à deux dimensions d’instances de type `Jeton` il faut quand même pouvoir dire qu’une case n’est pas occupée. Pour cela on peut rajouter une couleur supplémentaire que l’on appelle `VIDE` qui voudra dire que la case (i, j) est vide si la couleur de `donnees[i][j]` est `VIDE`.

Pour compléter les méthodes il nous faut avoir déjà défini la classe `Jeton`.

1.4.3 Le jeton

Nous allons coder la couleur du jeton dans un champ de type `int`. Si ce champ vaut 0 alors cela veut dire que la case est vide (cela correspond à la couleur `VIDE`). Si ce champ vaut 1 alors cela veut dire que le jeton est rouge. Si ce champ vaut 2 alors cela veut dire que le jeton break; est jaune.

Pour se souvenir de ces conventions nous allons créer les champs statiques constants `VIDE`, `ROUGE` et `JAUNE` pour éviter de se souvenir que `ROUGE` vaut 1 :

```

1  class Jeton {
2      public static final int VIDE = 0;
3      public static final int ROUGE = 1;
4      public static final int JAUNE = 2;
5      int couleur;
6      public Jeton() {
7          couleur = VIDE;
8      }
9      public Jeton(int c) {
10         couleur = c;
11     }
12     public int donneCouleur() {
13         return couleur;
14     }
15 }

```

On a créé un constructeur par défaut qui met la couleur à `VIDE`.

1.4.4 Le joueur humain

Pour l'humain le prototype de la fonction est :

```

1  class Humain {
2      Tableau tableau;
3      public Humain(Tableau t) {
4          tableau =t;
5      }
6      public void joue() {
7      }
8  }

```

1.4.5 Le joueur informatique

On retrouve un prototype similaire à l'humain pour l'ordinateur.

```

1  class Ordinateur {
2      Tableau tableau;
3      public Ordinateur(Tableau t) {
4          tableau =t;
5      }
6      public void joue() {
7      }
8  }

```

1.4.6 Et maintenant ?

Maintenant nous avons les objets il faut encore les lier entre eux : instancier un tableau, un joueur informatique, un joueur humain.

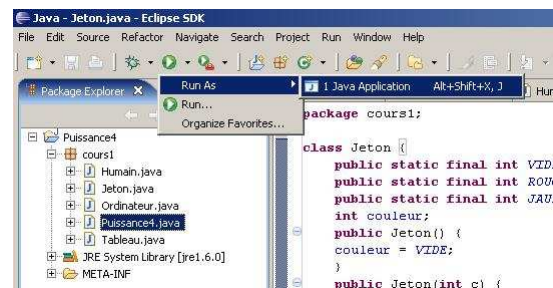
On peut faire cela dans une classe nommée `Puissance4` comportant une instance de `Tableau`, `Humain` et `Ordinateur`. On lui adjointra une méthode nommée `jouer` pour lancer le jeu. On mettra également la méthode `main` dans cette classe.

```

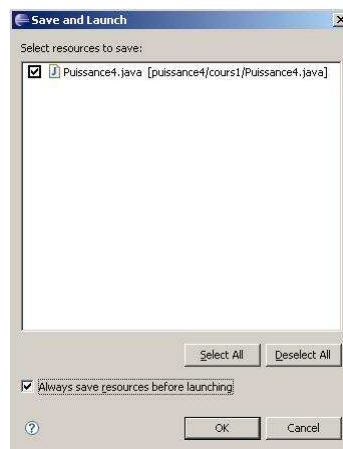
1  class Puissance4 {
2      Tableau tableau;
3      Humain humain;
4      Ordinateur ordinateur;
5      Puissance4() {
6      }
7      void jouer() {
8      }
9      public static void main(String argv[]) {
10     }
11 }

```

Une fois ces classes créées on peut les exécuter :



Sélectionner la classe Puissance4, sur le bouton “run” choisir “Run As” puis “Java Application”



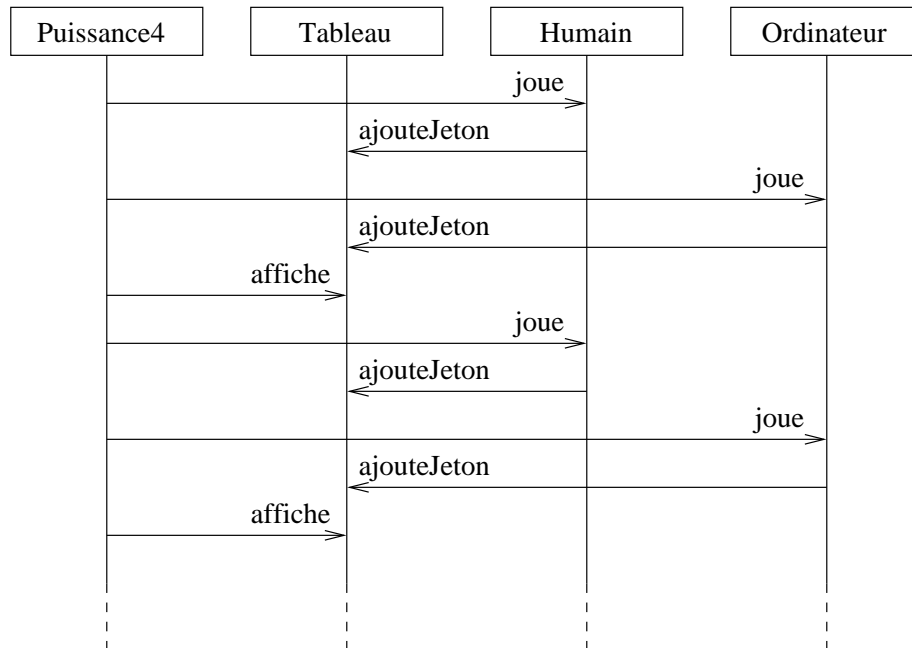
Choisir les ressources à sauver

1.5 Contenu des méthodes

Nous avons déterminé l’infrastructure de nos classes. Il faut maintenant remplir le corps des méthodes. Nous allons pour cela déterminer des “cas d’utilisation” (*use case* en anglais) permettant de donner des cas d’utilisation concrets.

1.5.1 Cas d’utilisation normal

Nous allons tout d’abord commencer par un cas simple : celui où une partie normale se déroule. Voici ci-dessous un diagramme où l’humain commence.



Déroulement d'une partie où l'humain commence.

C'est l'occasion de remarquer que nous avons oublié de dire comment la partie finissait. Il faut en effet rajouter un test après l'ajout d'un nouveau jeton pour savoir si un joueur a gagné ou bien s'il y a match nul. Nous ajoutons donc dans `Puissance4` la méthode indiquant si le jeu actuel est terminé :

```

boolean jeuTermine() { // à remplir
}
  
```

1.6 Conclusion

Dans le cours nous avons procédé aux rappels sur la description des objets dans le langage Java. Nous avons mis l'accent sur l'aspect compositionnel des objets à travers les attributs `public` et `private`.

Une méthodologie générique de la manière de concevoir les classes a été présentée à travers un exemple. En voici un rappel.

1. Déterminer approximativement un découpage en classes permettant de modéliser les données présentes.
2. Décrire textuellement les classes : leur rôle, les données conservées, la liste des actions à réaliser.
3. Ecrire le squelette du code Java des classes.
4. Remplir le corps des méthodes.

1.7 Exercices

Ex 1.1 * Nouvelles instances

On considère le programme Java ci-dessous :

```

1 class Ex1 {
2     int i=3;
3
4     Ex1() { }
  
```

```

5     int getValue() { return i;}
6     void setValue(int j) { i=j;}
7
8     static boolean methode1() {
9         Ex1 instance1 = new Ex1();
10        Ex1 instance2 = instance1;
11        instance2.i=4;
12        return instance1.i==3;
13    }
14    static boolean methode2() {
15        Ex1 instance1 = new Ex1();
16        Ex1 instance2 = new Ex1();
17        instance2 = instance1;
18        instance2.i=4;
19        return instance1.i==3;
20    }
21 }

```

Q1 Que retourne la méthode `Ex1.methode1`. Expliquez le résultat.

Q2 Que retourne la méthode `Ex1.methode2`. Expliquez le résultat.

Ex 1.2* Ordre d'initialisation

On a vu que l'on pouvait utiliser des champs statiques d'une classe sans qu'une instance ne soit créée. On en déduit donc que les champs statiques doivent être initialisés avant toute exécution de méthode d'une classe. Qu'affiche alors le programme ci-dessous :

```

1  class Ex2 {
2      static class MaSousClasse {
3          MaSousClasse(String s) {
4              System.out.println(s);
5          }
6      }
7      MaSousClasse champ1 = new MaSousClasse("Champ 1");
8      MaSousClasse champ2;
9      static MaSousClasse champ3 = new MaSousClasse("Champ 3");
10
11     Ex2() {
12         System.out.println("Constructeur de Ex2");
13         champ2 = new MaSousClasse("Champ 2");
14     }
15
16     public static void main(String[] argv) {
17         System.out.println("Execution de Ex2.main");
18         Ex2 ex2 = new Ex2();
19     }
20 }

```

Ex 1.3** Puissance 4

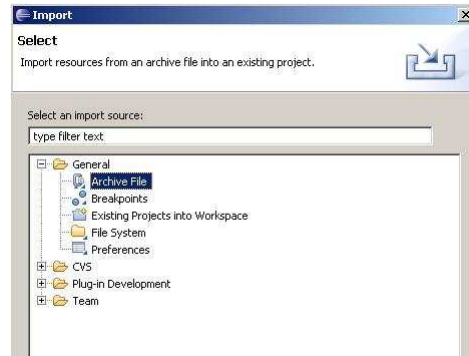
Le but de l'exercice est de remplir le corps des classes présentés dans ce cours pour disposer d'un jeu de Puissance 4.

Si vous êtes à l'aise faites-le vous-même. Sinon suivez les instructions ci-dessous :

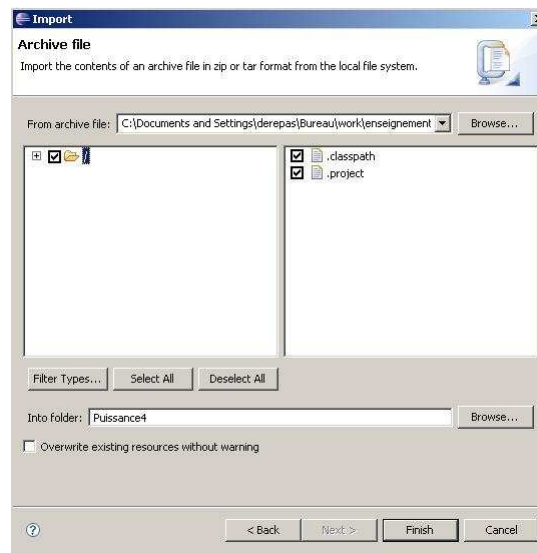
Q1 Téléchargez sur votre disque dur le fichier jar suivant :

http://www.derepas.com/java_int/ex_1_3.jar

Créer un nouveau projet comme indiqué à la section 1.4.1. Cliquer droit sur le nouveau projet et choisir "import"



Choisir “Archive File” puis cliquer sur suivant



Rentrer le nom du fichier jar précédemment téléchargé puis cliquer sur “Finish”, puis “Yes to all”

Q2 Compléter la méthode `joue` dans la classe `Ordinateur` pour jouer au hasard dans une colonne non pleine. Pour tirer un entier `i` et `j` entre 0 et 6 compris au hasard en Java on peut utiliser les instructions suivantes :

```
java.util.Random r = new java.util.Random();
int i = r.nextInt(7);
int j = r.nextInt(7);
```

Q3 faire de même avec la classe `Humain`

Q4 compléter la méthode `jouer` dans la classe `Puissance4`. Vérifier le résultat d’une partie.