

Programmation Orientée Objet JAVA

Compléments du cours - Exercices

Sommaire

Principes POO.....	2
1 Encapsulation.....	3
2 Syntaxe Java - compilation.....	4
3 État des objets.....	5
4 Référence vs objet.....	6
5 Constructeurs.....	8
6 Encapsulation.....	9
7 Enveloppes.....	11
8 Variable d'instance vs variable de classe (statique).....	12
9 Héritage.....	14
10 Constructeur et héritage.....	15
11 Polymorphisme.....	17
12 Redéfinition, sur-définition de méthode et polymorphisme.....	18
13 Tableau vs ArrayList.....	19
14 ArrayList.....	19
15 Mot croisé de synthèse.....	21
16 compareTo() vs compare().....	22
17 Première animation graphique.....	23
18 Écouteur d'événements souris.....	24
19 Classes internes.....	25
20 Graphique et événementiel.....	27
21 Et pour conclure.....	29

Extraits du livre : *Java - Tête la Première*. Kathy Sierra Bert Bates. 2005. O'Reilly.

Classe concernée : 3IRC – 4ETI

Date d'émission : juillet 2013

Émetteur : Françoise Perrin

Contact : fp@cpe.fr

Service : Info/Télécom – Bureau B123

Disponible sur : e-campus cours 4-6-COMSC4-C et
2-7-COMSC107-C

École Supérieure de Chimie Physique Électronique de Lyon
43, Bd du 11 novembre 1918 – Bât Hubert Curien - BP 2077
69616 Villeurbanne cedex – France
Tél. : +33 4 72 43 17 00 – Fax : +33 4 72 43 16 68 – www.cpe.fr

Principes POO

Qui suis-je ?



Un groupe de composants objets joue à « Qui suis-je ? ». Ils vous donnent des indices et vous devez essayer de deviner qui ils sont. On suppose qu'ils disent toujours la vérité. S'ils disent quelque chose qui peut être vrai de plusieurs d'entre eux, choisissez tous ceux auxquels l'assertion peut s'appliquer.

Les participants de ce soir : Classe, Méthode, Objet, Attributs.

1. Mes variables d'instance peuvent être différentes de celles de mes copains.
2. Je me comporte comme un patron.
3. J'aime faire des choses.
4. Je peux avoir plusieurs méthodes.
5. Je représente un « état ».
6. J'ai des comportements.
7. On me trouve dans des objets.
8. Je sers à créer des instances d'objets.
9. Mon état peut changer.
10. Je déclare des méthodes.
11. Je peux changer lors de l'exécution.

Les participants de ce soir : agrégat, classe agrégée, sous-classe, classe de base, classe composant.

1. On peut me substituer à mon type de base.
2. Mon comportement est utilisé comme une partie du comportement d'une autre classe.
3. Je change le comportement d'une autre classe.
4. Je ne change pas le comportement d'une autre classe.
5. Je donne mon comportement et ma fonctionnalité à mes sous-classes.
6. Je ne vais pas disparaître même si c'est le cas pour d'autres classes dans mes relations.
7. Je peux disparaître à l'insu de mon plein gré.
8. Je laisse quelqu'un d'autre faire des choses pour moi.

1 Encapsulation



L'interview de cette semaine : Un Objet nous parle franchement de l'encapsulation.



JTLP : Quel est vraiment l'intérêt de toute cette histoire d'encapsulation ?

Objet : Bien. Vous connaissez ce rêve dans lequel vous prononcez un discours devant cinq cents personnes pour vous rendre compte soudain que vous êtes nu ?

JTLP : Oui, ça m'est déjà arrivé. Ça me rappelle l'histoire du gars qui... Bon, passons. Alors vous vous sentez nu. Mais en dehors d'être un peu exposé, où est le danger ?

Objet : Où est le danger ? Où est le danger ? [il éclate de rire] Hé, les autres instances, vous avez entendu ça ? Il demande où est le danger ! [il se roule par terre]

JTLP : Qu'est-ce qu'il y a de drôle à ça ? Ça m'a l'air d'une question raisonnable.

Objet : Bon, Je vais vous expliquer. C'est... [il est repris d'un fou-rire incontrôlable].

JTLP : Donc l'encapsulation vous protège de quoi ?

Objet : L'encapsulation crée un champ de force autour de mes variables d'instance. Comme ça personne ne peut leur affecter quelque chose de, disons, impropre.

JTLP : Pouvez-vous me donner un exemple ?

Objet : Pas besoin d'un doctorat pour comprendre. Les valeurs de la plupart des variables d'instance doivent se situer dans un intervalle donné. Pensez à tout ce qui cesserait de fonctionner si les nombres négatifs étaient permis dans certaines situations. Le nombre de toilettes dans un bureau, la vitesse des avions, les anniversaires, les numéros de portables, la puissance des fours micro-ondes...

JTLP : Je vois. Et comment l'encapsulation permet-elle de fixer des bornes ?

Objet : En forçant le reste du code à passer par des méthodes set. Comme ça, la méthode set peut valider les paramètres et décider si c'est faisable. Elle peut rejeter la requête et ne rien faire, ou bien lancer une Exception (par exemple si un numéro de sécurité sociale a une valeur null), ou encore arrondir l'argument transmis à la plus proche valeur acceptable. L'important est que vous pouvez faire ce que vous voulez dans la méthode set, alors que vous ne pouvez rien faire si vos variables d'instance sont public.

JTLP : Mais je vois parfois des méthodes set qui définissent simplement une valeur sans rien tester. Si une variable d'instance n'a pas de bornes, est-ce que ça n'entraîne pas une surcharge inutile ? Une dégradation des performances ?

Objet : L'intérêt des méthodes set et get, **c'est que vous pouvez changer d'avis sans endommager le code de personne !** Imaginez que la moitié des employés de votre entreprise utilisent votre classe dont les variables d'instance sont publiques, et qu'un jour vous vous disiez soudain : « Zut ! Il y a quelque chose que je n'avais pas prévu avec cette valeur. Je vais devoir passer à une méthode set. » Le reste du code ne fonctionne plus. Le côté génial de l'encapsulation, c'est que personne n'est lésé si vous changez d'avis. Le gain de performance que permet l'utilisation directe des variables est si minuscule que le jeu n'en vaut pratiquement jamais la chandelle.

2 Syntaxe Java - compilation

Chaque fichier .java représente un fichier source complet. Ils ne se compilent ou ne s'exécute pas ou s'exécute mal, comment les corrigez-vous ?

1 Exercice1.java

```
public class Exercice1 {
    public static void main(String [] args) {
        int x = 1;
        while ( x < 10 ) {
            if ( x > 3) {
                System.out.println("grand x");
            }
        }
    }
}
```

2 Exercice2.java

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
        x = x - 1;
        if ( x < 3) {
            System.out.println("petit x");
        }
    }
}
```

3 Exercice3.java

```
public class Exercice3 {
    int x = 5;
    while ( x > 1 ) {
        x = x - 1;
        if ( x < 3) {
            System.out.println("petit x");
        }
    }
}
```

3 État des objets

La classe `FilmTestDrive` crée des objets (des instances) de la classe `Film` et utilise l'opérateur point (.) pour affecter une valeur spécifique aux variables d'instance. La classe `FilmTestDrive` invoque (appelle) une méthode sur l'un des objets.

1. Complétez les schémas avec les valeurs qu'ont ces trois objets à la fin de la méthode `main()`.
2. Quel est le titre du film sur lequel une méthode est appelée ?



```
class Film {
    String titre;
    String genre;
    int classement;
    public void leJouer() {
        System.out.println("Jouer le film");
    }
}

public class FilmTestDrive {
    public static void main(String[] args) {
        Film un = new Film();
        un.titre = "Autant en emporte la Bourse";
        un.genre = "Tragique";
        un.classement = -2;
        Film deux = new Film();
        deux.titre = "Perdus dans un box";
        deux.genre = "Comique";
        deux.classement = 5;
        deux.leJouer();
        Film trois = new Film();
        trois.titre = "Le club des octets";
        trois.genre = "Tragique mais édifiant";
        trois.classement = 127;
    }
}
```

titre
genre
classement

titre
genre
classement

titre
genre
classement

4 Référence vs objet



L'interview de cette semaine : Une référence nous livre ses secrets



JTLP : Eh bien, dites-moi, à quoi ressemble la vie d'une référence ?

Référence : C'est très simple, vraiment. Je suis une télécommande et on peut me programmer pour contrôler divers objets.

JTLP : Vous voulez dire plusieurs objets ? Par exemple référencer un Chien et cinq minutes après un Chat ?

Référence : Bien sûr que non. Une fois que j'ai été déclarée, si je suis une télécommande pour Chien, je ne pourrai jamais pointer sur... Aïe ! Pardon, nous ne sommes pas censés dire « pointer »... Je veux dire que je ne pourrai jamais référencer autre chose qu'un Chien.

JTLP : Cela veut-il dire que vous ne pouvez référencer qu'un seul Chien ?

Référence : Non. Je peux référencer un Chien et cinq minutes après un autre Chien. Tant que c'est un Chien, je peux être redirigée vers lui. A moins que... Non, laissez tomber.

JTLP : Non, dites-moi. Qu'alliez-vous dire ?

Référence : Je ne pense pas que vous vouliez entrer dans les détails maintenant, mais, pour abrégé, si je suis déclarée final, une fois que je suis liée à un Chien, on ne peut jamais me reprogrammer pour autre chose que ce seul et unique Chien. Autrement dit, on ne peut pas m'affecter d'autre objet Chien.

JTLP : Vous avez raison, ce n'est pas le moment d'en parler. Bien, donc, à moins d'être final, vous pouvez référencer un Chien donné, puis un autre Chien plus tard. Mais est-ce que vous pouvez ne rien référencer du tout ? Est-ce possible de n'être programmée pour rien ?

Référence : Oui, mais je n'ai pas envie d'en parler.

JTLP : Pourquoi donc ?

Référence : Parce que ça veut dire que je suis null, et ça me rend malade.

JTLP : Vous voulez dire que c'est parce que vous n'avez aucune valeur ?

Référence : Oh, null *est une valeur*. Je suis toujours une télécommande, mais c'est comme si vous rentriez à la maison avec une télécommande universelle et que vous n'aviez pas de télé. Je ne suis pas programmée pour contrôler quelque chose. Vous pouvez appuyer sur mes boutons toute la journée sans que rien ne se passe. Je me sens si... inutile. Un vrai gaspillage de bits. D'accord, pas tant de bits que ça, mais quand même. Et il y a pire. Si je suis la seule référence à un objet donné et qu'on m'affecte null (qu'on me déprogramme), plus personne ne peut accéder à l'objet que je référencais.

JTLP : Et où est le problème ?

Référence : Vous plaisantez ? J'ai développé une relation privilégiée avec un objet, une sorte d'intimité, et voilà que ce lien est brutalement et cruellement sectionné. Et je ne reverrai jamais cet objet, parce que maintenant il est bon pour le [musique tragique en arrière-fond] *ramasse-miettes*. Snif. Mais vous croyez que les programmeurs s'en soucient ? Snif. Pourquoi ne suis-je pas une primitive ? *J'ai horreur d'être une référence*. Toute cette responsabilité, ces amours brisées...

Dans l'exemple 3 :

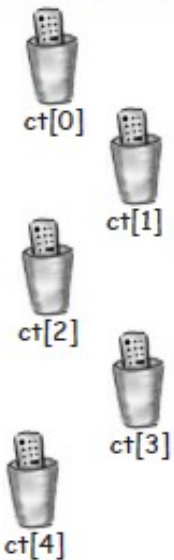
1. Combien d'objets sont créés ?
2. Combien de variables référencent chaque objet ?
3. Quel est le nom de la variable qui référence l'objet dont le genre est : "Tragique mais édifiant" ?
4. Combien d'espaces mémoire en tout sont-ils réservés en mémoire ? combien sont sur la pile, combien sur le tas ? illustrez par un dessin.



Reliez les variables du programme suivant aux objets correspondants

```
class CasseTete {
    int id = 0;
    public static void main(String [] args) {
        int x = 0;
        CasseTete [ ] ct = new CasseTete[5];
        while ( x < 3 ) {
            ct[x] = new CasseTete();
            ct[x].id = x;
            x = x + 1;
        }
        ct[3] = ct[1];
        ct[4] = ct[1];
        ct[3] = null;
        ct[4] = ct[0];
        ct[0] = ct[3];
        ct[3] = ct[2];
        ct[2] = ct[0];
        // suite du code
    }
}
```

Variables référence :



Objets CasseTete :



5 Constructeurs

Reliez l'appel de `new Canard()` avec le constructeur qui s'exécute quand chaque Canard est instancié.

```
public class TestCanard {
    public static void main(String[] args){
        int poids = 8;
        float densite = 2.3F;
        String nom = "Donald";
        long[] plumes = {1,2,3,4,5,6};
        boolean vole = true;
        int vitesse = 22;
        Canard[] c = new Canard[7];
        c[0] = new Canard();
        c[1] = new Canard(densite,poids);
        c[2] = new Canard(nom, plumes);
        c[3] = new Canard(vole);
        c[4] = new Canard(3.3F, vitesse);
        c[5] = new Canard(false);
        c[6] = new Canard(vitesse,densite);
    }
}
```

```
class Canard {
    int kilos = 3;
    float flottabilite = 2.1F;
    String nom = "Générique";
    long[] plumes = {1,2,3,4,5,6,7};
    boolean vole = true;
    int vitesseMax = 25;

    public Canard() {
        System.out.println("canard type1");
    }
    public Canard(boolean voler) {
        vole = voler;
        System.out.println("canard type2");
    }
    public Canard(String n, long[] f) {
        nom = n;
        plumes = f;
        System.out.println("canard type3");
    }
    public Canard(int k, float f) {
        kilos = k;
        flottabilite = f;
        System.out.println("canard type4");
    }
    public Canard(float densite,int max){
        flottabilite = densite;
        vitesseMax = max;
        System.out.println("canard type5");
    }
}
```


6 Encapsulation

Qui suis-je ?



Un groupe de composants Java joue à « Qui suis-je ? ». Ils vous donnent des indices et vous devez essayer de deviner qui ils sont. On suppose qu'ils disent toujours la vérité. S'ils disent quelque chose qui peut être vrai de plusieurs d'entre eux, choisissez tous ceux auxquels l'assertion peut s'appliquer.

Les participants de ce soir : variable d'instance, argument, retour, méthode get, méthode set, encapsulation, public, private, passage par valeur, méthode

- 1 Une classe peut en avoir un nombre quelconque.
- 2 Une méthode ne peut en avoir qu'un.
- 3 Peut être converti implicitement.
- 4 Je préfère mes variables d'instance private.
- 5 Signifie réellement « faire une copie ».
- 6 Seules les méthodes set devraient les modifier.
- 7 Une méthode peut en avoir plusieurs.
- 8 Je retourne quelque chose par définition.
- 9 Ne m'utilisez pas avec des variables d'instance.
- 10 Je peux avoir plusieurs arguments.
- 11 Par définition, j'accepte un seul argument.
- 12 Ils aident à créer l'encapsulation.
- 13 Je vole toujours en solo.

Donner la trace d'exécution du programme suivant :

```
class Joueur {
    private boolean possessionDeLaBalle;
    private int cartonJaune;
    private boolean expulsion;
    Joueur() {
        possessionDeLaBalle = false;
        cartonJaune = 0;
        expulsion = false;
    }
    public void recevoirLaBalle() {
        possessionDeLaBalle = true;
    }
    public void recevoirCarton() {
        cartonJaune++;
    }
    public void perdreLaBalle() {
        possessionDeLaBalle = false;
    }
    public boolean getPossession() {
        return possessionDeLaBalle;
    }
    public int getNombreCartons() {
        return cartonJaune;
    }
    public void expulsion() {
        expulsion = true;
    }
}
```

```

    }
    public boolean getExpulsion() {
        return expulsion;
    }
}

public class Football {

    public static void main(String[] argv) {
        Joueur[] joueurs = new Joueur[5];
        for(int i=0; i<5; i++){
            joueurs[i] = new Joueur();
        }
        joueurs[2].recevoirLaBalle();
        faireUnePasse(joueurs[2], joueurs[0]);
        quiALaBalle(joueurs);
        faireUnePasse(joueurs[0], joueurs[4]);
        quiALaBalle(joueurs);
        joueurs[3].recevoirCarton();
        checkExpulsion(joueurs[3]);
        faireUnePasse(joueurs[3], joueurs[2]);
        joueurs[3].recevoirCarton();
        checkExpulsion(joueurs[3]);
        faireUnePasse(joueurs[4], joueurs[3]);
    }
    public static void faireUnePasse(Joueur a, Joueur b) {
        if (a.getPossession()) {
            if (!b.getExpulsion()) {
                a.perdreLaBalle();
                b.recevoirLaBalle();
            }
            else {
                System.out.println("Le joueur ne peut recevoir la balle car il est
                    expulse.");
            }
        }
        else {
            System.out.println("Ce joueur ne possede pas le ballon.");
        }
    }
    public static void checkExpulsion(Joueur a) {
        if (a.getNombreCartons() == 2) {
            a.expulsion();
        }
    }
    public static void quiALaBalle(Joueur[] a) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].getPossession()) {
                System.out.println("C'est le joueur numero " + (i+1) + " qui a la
                    balle.");
                break;
            }
        }
    }
}

```

7 Enveloppes,

Vrai ou Faux ?

1. Il n'y a pas de classe enveloppe pour les booléens.
2. On utilise une enveloppe pour traiter une valeur primitive comme un objet.
3. Les méthodes parseXxx retournent toujours une chaîne de caractères.

Le code suivant va-t-il compiler, s'exécuter, pourquoi ?

```
public class TestBoxing {
    Integer i;
    int j;
    public static void main (String[] args) {
        TestBoxing t = new TestBoxing();
        t.go();
    }
    public void go() {
        j=i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

8 Variable d'instance vs variable de classe (statique)

Face à Face



Variable d'instance

Je ne sais même pas à quoi sert cette discussion. Tout le monde sait que les variables statiques ne servent que pour les constantes. Et combien y en a-t-il ? Je pense que toute l'API doit en avoir... combien ? Quatre ? Et on ne les utilise pas tous les jours.

Pleine. Vous pouvez répéter ? O.K., il y en a un peu dans la bibliothèque Swing, mais tout le monde sait que Swing est un cas à part.

O.K., mais en dehors de trois ou quatre trucs de ce genre, donnez-moi un exemple d'une seule variable statique que tout le monde puisse utiliser. Dans le monde réel, je veux dire.

Mais c'est un autre cas à part. Et personne ne l'utilise de toute façon, sauf pour le débogage.

Hum, est-ce que vous n'oubliez rien ?

Il n'y a rien de moins OO qu'une variable statique !! Pourquoi ne pas faire un gigantesque pas en arrière et faire de la programmation procédurale pendant qu'on y est ?

Vous êtes comme une variable globale, et tout programmeur qui se respecte sait que c'est généralement une Mauvaise Chose.

Variable statique

Vous devriez vérifier vos sources. Quand avez-vous donc consulté l'API pour la dernière fois ? Elle est bourrée à craquer de statiques ! Elle a même des classes entières destinées à contenir des valeurs constantes. Par exemple, il y a une classe `SwingConstants` qui en est carrément pleine.

C'est peut-être un cas à part, mais il est vraiment important ! Et que dites-vous de la classe `Color` ? Quelle galère si on devait mémoriser toutes les valeurs RGB pour obtenir les couleurs standard ! Heureusement, la classe `Color` a déjà des constantes définies pour bleu, jaune, blanc, rouge, etc. Très pratique.

Eh bien `System.out` pour commencer ? Le `out` de `System.out` est une variable statique de la classe `System`. Vous ne créez pas personnellement une nouvelle instance de `System`, vous demandez simplement à `System` sa variable `out`.

Oh, parce que le débogage n'est pas important ?

Et autre chose qui n'a pas dû traverser votre petite tête : les variables statiques sont plus efficaces. Une par classe au lieu d'une par instance. Les économies de mémoire pourraient être énormes !

Quoi ?

Que voulez-vous dire par rien de moins OO ?

Je ne suis PAS une variable globale. Ça n'existe pas. Je vis dans une classe ! C'est Oui, vous vivez dans une classe. Mais on n'appelle pas ça de la programmation orientée objet. C'est tout simplement stupide. Vous êtes une relique. Quelque chose qui aide les anciens programmeurs à s'adapter à Java.

Bon, O.K., de temps à autre sans doute, cela a un sens d'employer une variable statique. Mais laissez-moi vous dire que l'abus des variables (et des méthodes) statiques est la marque d'un programmeur OO immature. Un bon concepteur doit penser à l'état des *objets*, pas à celui des *classes*.

Les méthodes statiques, c'est le pire de tout, parce qu'elles signifient généralement que le programmeur a une pensée procédurale au lieu de voir les objets agissant en fonction de leur état d'objet unique.

Bon... Si vous voulez vous raconter des histoires...

Vrai ou Faux ?

1. Pour utiliser la classe Math, il faut d'abord en créer une instance.
2. On peut marquer un constructeur avec le mot-clé **static**.
3. Les variables statiques n'ont pas accès à l'état des variables d'instance de l'objet « this ».
4. C'est une bonne pratique d'appeler une méthode statique avec une variable référence.
5. Les variables statiques peuvent servir à compter les instances d'une classe.
6. Les constructeurs sont appelés avant que les variables statiques ne soient initialisées.
7. MAX_SIZE serait un bon nom pour une variable statique finale.
8. Un bloc initialiseur statique s'exécute avant un constructeur de la classe.
9. Si une classe est finale, toutes ses méthodes doivent être finales.

parfaitement OO vous savez, une CLASSE. Je ne suis pas suspendue dans l'espace comme ça : je fais naturellement partie de l'état d'un objet. La seule différence, c'est que je suis partagée par toutes les instances d'une classe. C'est très efficace.

Arrêtez-vous tout de suite. C'est complètement FAUX. Certaines variables statiques sont absolument capitales pour un système. Et même celles qui ne sont pas capitales sont pratiques.

Pourquoi dites-vous ça ? Et qu'est-ce que vous reprochez aux méthodes statiques ?

Bien sûr, je sais que les objets doivent être au centre de la conception OO, mais ce n'est pas parce qu'il y a quelques programmeurs ignorants qu'il faut jeter le bébé avec le bytecode. Il y a un temps et un lieu pour les variables statiques, et quand vous en avez besoin elles sont imbattables.

9 Héritage

Soit le programme suivant :

```
public class Docteur {
    protected boolean travailleALHopital;
    public void traiterPatient() {
        // faire un checkup
    }
}

public class MedecinDeFamille extends Docteur {
    private boolean faitDesVisites;
    public void donnerConseil() {
        // donner un simple conseil
    }
}

public class Chirurgien extends Docteur{
    public void traiterPatient() {
        // opérer
    }
    public void faireUneIncision() {
        // inciser (aïe !)
        // utilisée lors d'une opération uniquement
    }
}
```

- 1 Quel est le nombre de variables d'instance de Chirurgien ?
- 2 Quel est le nombre de variables d'instance de MedecinDeFamille ?
- 3 Combien de méthodes Docteur a-t-il ?
- 4 Combien de méthodes Chirurgien a-t-il ?
- 5 Combien de méthodes MedecinDeFamille a-t-il ?
- 6 MedecinDeFamille peut-il traiterPatient()?
- 7 Peut-il faireUneIncision() ?
- 8 Quelle méthode devrait être privée ?
- 9 Comment pourrions-nous introduire des interfaces dans cette organisation ?

Il y a 2 interfaces à trouver.

La première contient deux méthodes (une facile, une plus difficile).

La deuxième contient une seule méthode.

10 Constructeur et héritage

1 Soit le programme suivant :

```
public class BacASable {
    public static void main(String[] args) {
        B b1 = new B();
        B b2 = new B(2003);
        B b3 = new B("bonjour");
        System.out.println(b1.getx( ) + " et " + b2.getx( ) + " et encore " +
b3.getx( ) );
    }
}

class A
{
    public int x;
    public A( ) {
        x=5;
    }
    int getx( ) {
        return x;
    }
}

class B extends A
{
    public B( ) {
        x++;
    }
    public B(int i) {
        this( );
        x=x+i;
    }
    public B(String s) {
        super();
        x--;
    }
}
}
```

Quelle est la bonne trace d'exécution ?

- (a) 6 et 2009 et encore 4
- (b) 1 et 2004 et encore 4
- (c) 1 et 2004 et encore 2003
- (d) une erreur d'exécution

2 Soient les classes Boo et SonOfBoo.

```
public class Boo {
    public Boo(int i) { }
    public Boo(String s) { }
    public Boo(String s, int i) { }
}

class SonOfBoo extends Boo {
    public SonOfBoo() {
        super("boo");
    }
    public SonOfBoo(int i) {
        super("Fred");
    }
    public SonOfBoo(String s) {
        super(42);
    }
    public SonOfBoo(int i, String s) {
    }
    public SonOfBoo(String a, String b, String c){
        super(a,b);
    }
    public SonOfBoo(int i, int j) {
        super("man", j);
    }
    public SonOfBoo(int i, int x, int y) {
        super(i, "star");
    }
}
```

Quels sont les constructeurs de la classe SonOfBoo qui sont illégaux ?

11 Polymorphisme

Soit le programme suivant :

```
public class Animaux {
    public static void main(String[] args) {
        Animal titi = new Canari(3);
        titi.vieillir();
        titi.vieillir(2);
        titi.crier();
    }
}

abstract class Animal {
    boolean vivant;
    int âge;
    int âgeMaxi = 10;
    Animal() {
        this(1);
    }
    Animal (int a) {
        âge = a;
        vivant = true;
        System.out.println("Un animal de " + a + " an(s) vient d'être créé");
    }
    abstract void crier();

    void vieillir() {
        vieillir(1);
    }
    void vieillir(int a) {
        âge += a;
        afficheAge();
        if (âge > âgeMaxi) {
            mourir();
        }
    }
    private void afficheAge() {
        System.out.println("C'est l'anniversaire de cet animal.");
        System.out.println("Il a maintenant " + âge + " an(s)");
    }
    private void mourir() {
        vivant = false;
        System.out.println("Cet animal est mort");
    }
}

class Canari extends Animal {
    Canari(int a) {
        super(a);
    }
    void crier() {
        System.out.println("Cui-cui !");
    }
}
```

- 1 Donnez la trace d'exécution du programme.
- 2 Pourquoi la classe Animal doit-elle être abstraite ?
- 3 Si l'on remplace l'instruction `Animal titi = new Canari(3);` par l'instruction `Animal titi = new Canari();` le programme ne fonctionne plus. Pourquoi ? Quelle correction apporter ?

12 Redéfinition, sur-définition de méthode et polymorphisme

- 1 Donner la trace d'exécution du programme suivant :

```
class A { public void affiche() { System.out.print ("Je suis un A ") ; }}
class B extends A {}
class C extends A { public void affiche() { System.out.print("Je suis un C ");}}
class D extends C { public void affiche() { System.out.print("Je suis un D ");}}
class E extends B {}
class F extends C {}
public class Poly {
    public static void main (String arg[]){
        A a = new A() ; a.affiche() ; System.out.println() ;
        B b = new B() ; b.affiche() ;
        a = b ; a.affiche() ; System.out.println() ;
        C c = new C() ; c.affiche() ;
        a = c ; a.affiche() ; System.out.println() ;
        D d = new D() ; d.affiche() ;
        a = d ; a.affiche() ;
        c = d ; c.affiche() ; System.out.println() ;
        E e = new E() ; e.affiche() ;
        a = e ; a.affiche() ;
        b = e ; b.affiche() ; System.out.println() ;
        F f = new F() ; f.affiche() ;
        a = f ; a.affiche() ;
        c = f ; c.affiche() ;
    }
}
```

- 2 Donner la trace d'exécution du programme suivant :

```
class AA {
    public void f(double x) { System.out.print ("A.f(double=" + x + ") ") ; }
}
class BB extends AA {}
class CC extends AA{
    public void f(long q) { System.out.print ("C.f(long=" + q + ") ") ; }
}
class DD extends CC{
    public void f(int n) { System.out.print ("D.f(int=" + n + ") ") ; }
}
class FF extends CC{
    public void f(float x) { System.out.print ("F.f(float=" + x + ") ") ; }
    public void f(int n) { System.out.print ("F.f(int=" + n + ") ") ; }
}
public class PolySur{
    public static void main (String arg[]){
        byte bb=1; short p=2; int n=3; long q=4;
        float x=5.5f ; double y=6. ;
        System.out.println ("** A ** ") ;
        AA a = new AA(); a.f(bb); a.f(x); System.out.println();
        System.out.println ("** B ** ") ;
        BB b = new BB(); b.f(bb); b.f(x); System.out.println();
        a = b; a.f(bb); a.f(x); System.out.println();
        System.out.println ("** C ** ") ;
        CC c = new CC(); c.f(bb); c.f(q); c.f(x); System.out.println();
        a = c; a.f(bb); a.f(q); a.f(x); System.out.println();
        System.out.println ("** D ** ") ;
        DD d = new DD(); d.f(bb); c.f(q); c.f(y); System.out.println();
        a = c; a.f(bb); a.f(q); a.f(y); System.out.println();
        System.out.println ("** F ** ") ;
        FF f = new FF(); f.f(bb); f.f(n); f.f(x); f.f(y); System.out.println();
        a = f; a.f(bb); a.f(n); a.f(x); a.f(y); System.out.println();
        c = f; c.f(bb); c.f(n); c.f(x); c.f(y); System.out.println();
    }
}
```

13 Tableau vs ArrayList



L'interview de cette semaine : Une ArrayList persifle sur les tableaux.



JTLP : Donc les ArrayList sont des tableaux, n'est-ce pas ?

ArrayList : Dans leurs rêves ! **Je suis un objet**, merci bien.

JTLP : Si ma mémoire est bonne, les tableaux aussi. Ils vivent sur le tas, avec tous les autres objets.

ArrayList : Bien sûr qu'ils vont sur le tas. Mais un tableau est toujours un ArrayList raté. Un poseur. Les objets ont un état et un comportement, nous sommes d'accord ? Parfait. Mais avez-vous déjà essayé d'appeler une méthode sur un tableau ?

JTLP : Maintenant que vous en parlez, je ne peux pas dire que j'ai essayé. Mais qu'est-ce que j'aurais bien pu appeler comme méthode ? J'appelle des méthodes sur ce que j'ai mis dans le tableau, pas sur le tableau lui-même. Et la syntaxe des tableaux me permet d'y mettre des choses et d'en retirer.

ArrayList : Vraiment ? Vous essayez de me dire que vous avez déjà vraiment retiré quelque chose d'un tableau ? (Pfff... Où est-ce que vous avez tous été formés ? Chez McJava ?)

JTLP : Bien sûr que je peux retirer quelque chose d'un tableau. Je dis `Chien c = tableauDeChiens[1]` et j'extrais l'objet Chien qui occupe l'indice 1 du tableau.

ArrayList : Très bien, je vais parler lentement pour que vous puissiez me suivre. Vous n'avez pas, je répète, pas retiré ce Chien du tableau. Tout ce que vous avez fait, c'est copier la référence au Chien et l'affecter à une autre variable Chien.

JTLP : Oh, je vois ce que vous voulez dire. Je n'ai pas retiré physiquement le Chien. Il est toujours là. Mais je peux toujours mettre sa référence à null, je suppose.

ArrayList : Mais, voyez-vous, je suis un objet de première classe. J'ai donc des méthodes, et je peux vraiment supprimer l'objet Chien au lieu de me contenter de lui affecter null. Et je peux changer de taille dynamiquement. Allez demander à un tableau de faire ça !

JTLP : Euh... ça me gêne d'aborder ce sujet, mais le bruit court que vous n'êtes rien d'autre qu'un vulgaire tableau, en moins efficace. Que vous n'êtes en fait qu'une enveloppe pour un tableau, avec des méthodes supplémentaires comme le redimensionnement et tout ça que je pourrais écrire moi-même. Et pendant que j'y suis, vous ne pouvez même pas contenir de types primitifs ! Est-ce que ce n'est pas une limitation importante ?

ArrayList : Je n'arrive pas à croire que vous vous laissiez abuser par cette légende urbaine. Non, je ne suis pas qu'un tableau en moins efficace. J'admets qu'il y a quelques situations extrêmement rares dans lesquelles un tableau pourrait être juste un poil, je répète, un poil plus rapide pour certaines choses. Mais ce minuscule gain en performance justifie-t-il l'abandon de toute cette puissance ? Et de toute cette souplesse ? Et en ce qui concerne les types primitifs, bien sûr que vous pouvez placer un type primitif dans une ArrayList, tant que vous l'emballez dans une classe enveloppe. Et depuis Java 5.0, cette opération (et l'opération inverse quand vous récupérez le type primitif) a lieu automatiquement. Et, d'accord, je reconnais que oui, si vous utilisez une ArrayList de types primitifs, un tableau est probablement plus rapide, à cause de tout l'emballage et le débballage, mais quand même... Cet infime gain de performance en vaut-il la peine ?

14 ArrayList

Le programme suivant a été découpé en morceaux collés avec des magnets sur le frigo.



Pouvez-vous réorganiser les fragments de code pour obtenir un programme Java qui fonctionne et affiche le résultat suivant :

```
zéro un deux trois
zéro un trois quatre
zéro un trois quatre 4.2
zéro un trois quatre 4.2
```

Indices

La classe s'appelle **ArrayListMagnet**

Elle manipule un attribut de type **ArrayList<String>**

La fonction **main()** fait appel à une fonction **printAL()** pour afficher les valeurs de la liste à l'écran. Sa signature est la suivante :

```
public static void printAL(ArrayList<String> al)
```

```
a.remove(2);
```

```
printAL(a);
```

```
a.add(0, "zéro");  
a.add(1, "un");
```

```
printAL(a);
```

```
public static void printAL(ArrayList<String> al) {
```

```
    if (a.contains("deux")) {  
        a.add("2.2");  
    }
```

```
    a.add(2, "deux");
```

```
public static void main (String[] args) {
```

```
    System.out.print(element + " ");  
}  
    System.out.println(" ");
```

```
    if (a.contains("trois")) {  
        a.add("quatre");  
    }
```

```
public class ArrayListMagnet {
```

```
    if (a.indexOf("quatre") != 4) {  
        a.add(4, "4.2");  
    }
```

```
)
```

```
)
```

```
import java.util.*;
```

```
printAL(a);
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
for (String element : al) {
```

```
    a.add(3, "trois");  
    printAL(a);
```

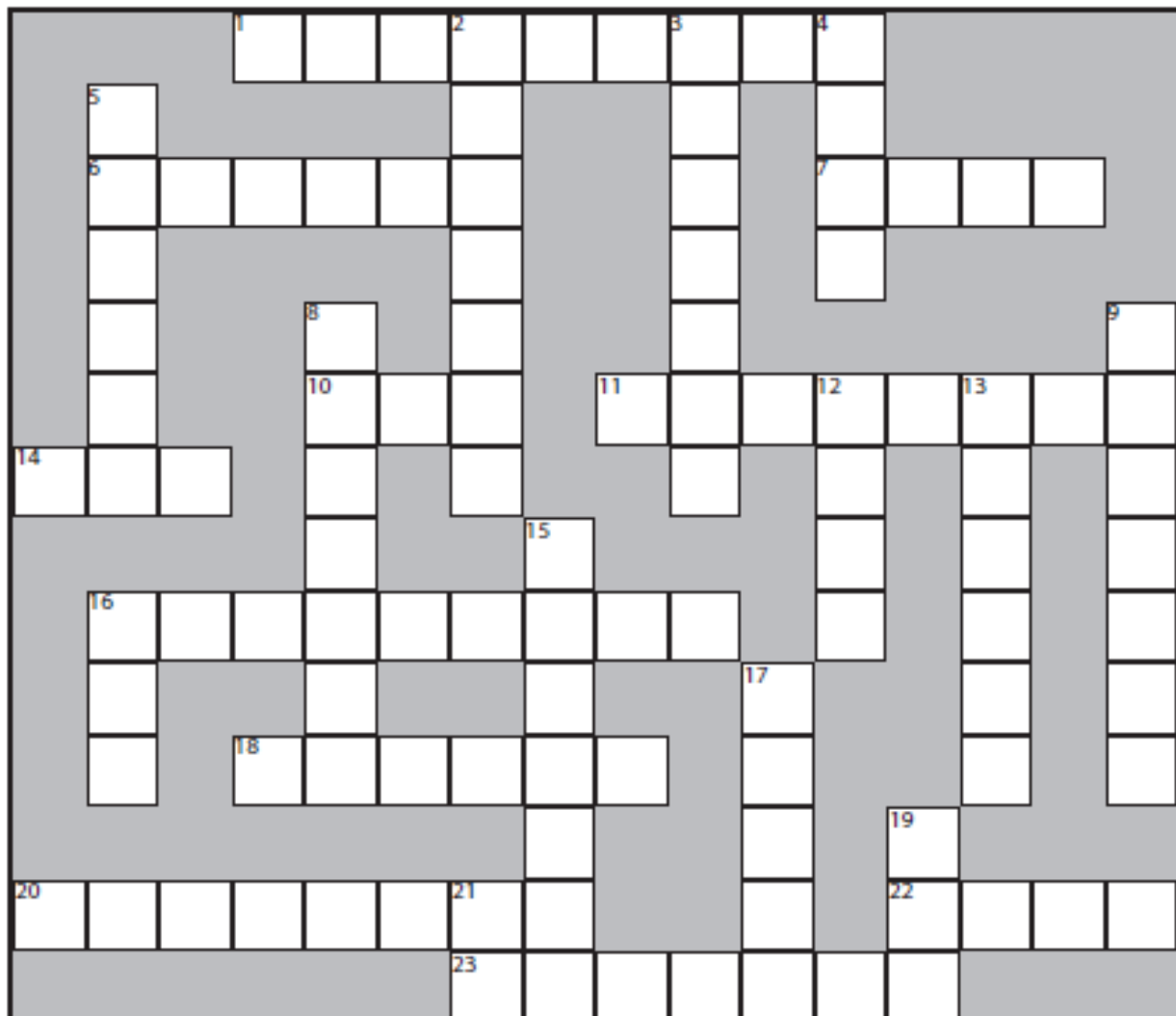
15 Mot croisé de synthèse

Horizontalement

1. Types qui n'ont pas de comportement
6. Tous les objets en dérivent
7. Premier indice
10. Ajoute un objet
11. Loup y es-tu (dans une collection) ?
14. Entrée interdite aux doublons
16. Evolue avec la croissance
18. Homologue de 4
20. Inconstante
22. Sinon
23. Unité adressable

Verticalement

2. Implémente un comportement
3. Retourne une position
4. Retourne un nombre d'éléments.
5. Grand décimal
8. Groupe des classes
9. Pour savoir si une liste est vide
12. une variable a un nom et un ____
13. Position d'un élément
15. Pas réel
16. Acronyme de bibliothèque
17. Commun aux courses et aux tableaux
19. Récupère une valeur
21. Article (rien à voir avec Java)



16 compareTo() vs compare()

Pour chacune des questions ci-dessous, remplissez le blanc avec l'un des mots de la liste des réponses possibles.

Réponses possibles :

Comparator,
Comparable,
compareTo(),
compare(),
oui,
non

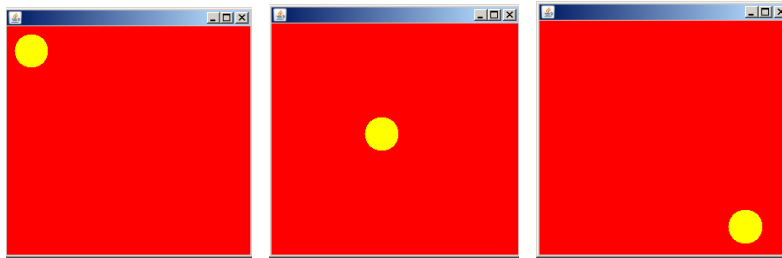
Étant donné l'instruction compilable suivante : `Collections.sort(monArrayList)` ;

1. Que doit implémenter la classe des objets stockés dans monArrayList ? _____
2. Quelle méthode la classe des objets stockés dans monArrayList doit-elle implémenter ? _____
3. La classe des objets stockés dans mon monArrayList peut-elle implémenter à la fois Comparator ET Comparable? _____

Étant donné l'instruction compilable suivante : `Collections.sort(monArrayList, maComparaison)` ;

4. La classe des objets stockés dans monArrayList peut-elle implémenter Comparable ? _____
5. La classe des objets stockés dans monArrayList peut-elle implémenter Comparator ? _____
6. La classe des objets stockés dans monArrayList doit-elle implémenter Comparable ? _____
7. La classe des objets stockés dans monArrayList doit-elle implémenter Comparator ? _____
8. Que doit implémenter la classe de l'objet maComparaison ? _____
9. Quelle méthode la classe de l'objet maComparaison doit-elle implémenter ? _____

17 Première animation graphique



Le but du programme suivant est de créer une animation simple, dans laquelle un disque va traverser l'écran, de bas en haut et de gauche à droite.

```
import javax.swing.*;
import java.awt.*;

public class SimpleAnimation {

    JFrame cadre;
    JPanel panneau;

    public static void main (String[] args) {
        SimpleAnimation ihm = new SimpleAnimation ();
        ihm.go();
    }

    public void go() {
        int x = 0;
        int y = 0;

        cadre = new JFrame();
        cadre.setSize(300,300);
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panneau = new Panneau();
        cadre.getContentPane().add(panneau,"Center");
        cadre.setVisible(true);

        // boucle pour afficher un disque jaune qui se déplace
        // dans le panneau rouge
        for (int i = 0; i < 220; i++) {
            x++;
            ((Panneau) panneau).setX(x);
            y++;
            ((Panneau) panneau).setY(y);
            panneau.repaint();

            // temporisation pour que l'utilisateur ait le temps
            // de voir se déplacer le disque
            try {
                Thread.sleep(30);
            } catch (Exception ex) { }
        }
    }
}

class Panneau extends JPanel {

    // ToDo
}
```

1. Ecrivez la classe Panneau nécessaire à la bonne exécution du programme.
2. Pourquoi un transtypage (cast) est-il nécessaire dans l'instruction :

```
((Panneau) panneau).setX(x);
```

18 Écouteur d'événements souris

Ce programme affiche un bouton dans une fenêtre. Ce bouton devrait changer de couleur en fonction des événements souris.

```
public class BoutonColoreDS {
    public static void main (String args[]) {
        new Fenetre().setVisible(true) ;
    }
}

class Fenetre extends JFrame{
    private JPanel pan = new JPanel();
    private JButton bouton = new Bouton("Mon bouton", Color.blue);
    public Fenetre(){
        this.setTitle("Animation");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pan.add(bouton);
        this.setContentPane(pan);
    }
}

class Bouton extends JButton implements MouseListener{
    private Color color;
    public Bouton(String str, Color color){
        super(str);
        this.color = color;
    }
    public void paintComponent(Graphics g){
        super.paintComponent(g);
        this.setBackground(color);
    }
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {
        color = Color.yellow; // fond en jaune lors du survol
        repaint();
    }
    public void mouseExited(MouseEvent event) {
        color = Color.green; // fond en vert lorsqu'on quitte le bouton
        repaint();
    }
    public void mousePressed(MouseEvent event) {
        color = Color.red; // fond en rouge lors du clic gauche
        repaint();
    }
    public void mouseReleased(MouseEvent event) {
        color = Color.blue; // fond en bleu lorsqu'on relâche le clic
        repaint();
    }
}
```

1. Pourquoi le résultat n'est pas celui attendu ?
2. Apportez la correction nécessaire au bon endroit.
3. Pourquoi faut-il écrire à vide la méthode `mouseClicked()` ?

19 Classes internes



L'interview de cette semaine : Une classe interne nous parle de ses relations avec sa classe externe.



JTLP : Quel est l'intérêt des classes internes ?

Objet interne : Par où commencer ? Nous vous donnons une chance d'implémenter la même interface plusieurs fois dans une classe. Vous vous souvenez qu'une classe Java normale ne permet pas d'implémenter une méthode plus d'une fois. Mais comme chaque classe interne peut implémenter la même interface, vous pouvez avoir plein d'implémentations différentes des mêmes méthodes d'une interface.

JTLP : Et pourquoi voudrait-on implémenter la même méthode deux fois ?

Objet interne : Revoyons les gestionnaires d'événements des IHM. Réfléchissez... Si vous voulez que trois boutons aient chacun un comportement différent, vous utiliserez trois classes internes qui implémenteront toutes trois ActionListener — ce qui signifie que chaque classe implémentera sa propre méthode actionPerformed().

JTLP : Les gestionnaires d'événements sont donc la seule raison d'utiliser des classes internes ?

Objet interne : Bien sûr que non. Les gestionnaires d'événements ne sont qu'un exemple évident. Chaque fois que vous avez besoin d'une classe séparée, mais que vous voulez quand même que cette classe se comporte comme si elle faisait partie d'une autre classe, une classe interne est la meilleure solution et parfois la seule.

JTLP : Il y a encore quelque chose que je ne comprends pas. Si vous voulez que la classe interne se comporte comme si elle appartenait à la classe externe, pourquoi avoir une classe séparée ? Pourquoi le code de la classe interne ne serait-il pas dans la classe externe ?

Objet interne : Je viens de vous citer un scénario dans lequel vous avez besoin de plusieurs implémentations d'une même interface. Mais même si vous n'utilisez pas d'interfaces, vous pouvez avoir besoin de deux classes différentes parce que ces deux classes représentent deux entités différentes. C'est de la bonne approche objet.

JTLP : Stop. Je croyais qu'une bonne partie de la conception OO concernait la réutilisation et la maintenance. Vous savez, l'idée selon laquelle si on a deux classes séparées, on peut les modifier et les utiliser indépendamment, au lieu de tout fourrer dans une seule classe, etc. Mais avec une classe interne, il n'y a jamais qu'une seule classe à la fin, non ? La classe externe est la seule qui soit réutilisable et séparée de tout le reste. Les classes internes ne sont pas précisément réutilisables. En fait, j'ai entendu dire qu'on les qualifiait de « réinutilisables ».

Objet interne : Oui, il est vrai qu'une classe interne n'est pas aussi réutilisable, quelquefois même pas du tout, parce qu'elle est intimement liée aux variables d'instance et aux méthodes de la classe externe.

JTLP : Ce qui ne fait qu'apporter de l'eau à mon moulin ! Si elle n'est pas réutilisable, pourquoi se soucier de créer une classe séparée. Je veux dire en dehors du problème des interfaces, qui m'a plutôt l'air d'un expédient.

Objet interne : Comme je vous l'ai dit, vous devez penser au polymorphisme.

JTLP : O.K. Et pourquoi penserais-je au polymorphisme ?

Objet interne : Parce que les classes externes et internes peuvent devoir être de types différents ! Commençons par l'exemple de l'écouteur polymorphe d'une IHM. De quel type est l'argument de la méthode qui enregistre l'écouteur du bouton ? Autrement dit, si vous vérifiez dans la documentation, de quel type (classe ou interface) est l'argument que vous passez à la méthode addActionListener() ?

JTLP : Vous devez passer un écouteur. Quelque chose qui implémente une interface écouteur particulière, en l'occurrence ActionListener. Tout le monde sait ça. Où voulez-vous en venir ?

Objet interne : Je veux en venir au fait que, du point de vue du polymorphisme, vous avez une méthode qui n'accepte qu'un type particulier. Quelque chose qui EST-UN ActionListener. Mais — et c'est là le point important — que se passe-t-il si votre classe doit avoir une relation EST-UN avec un type de classe et non d'interface ?

JTLP : Vous ne pouvez pas juste faire dériver votre classe de celle dont elle doit être une partie ? Est-ce que ce n'est pas justement tout l'intérêt du sous-classement ? Si B est une sous-classe de A, on peut utiliser un B partout où on attend un A ? Toute cette histoire de passer un Chien là où un Animal est le type déclaré ?

Objet interne : Bingo ! Et si vous devez réussir le test EST-UN pour deux classes différentes ? Des classes qui ne sont pas dans la même hiérarchie d'héritage ?

JTLP : Oh, eh bien, on peut juste... hmmm. Je crois que j'ai compris. On peut toujours implémenter plusieurs interfaces, mais on ne peut étendre qu'une seule classe. Vous ne pouvez réussir qu'un seul test EST-UN quand il s'agit de types de classes.

Objet interne : Exactement ! Vous ne pouvez pas être à la fois un Chien et un Bouton. Mais si vous êtes un Chien qui a parfois besoin d'être un Bouton (pour pouvoir vous transmettre vous-même à des méthodes qui acceptent un Bouton), la classe Chien (qui étend Animal et ne peut donc étendre Bouton) peut avoir une classe interne qui agit en tant que Bouton de la part du Chien en étendant la classe Bouton. Et chaque fois qu'un Bouton est requis, le Chien peut transmettre son Bouton interne au lieu de se transmettre lui-même. Autrement dit, au lieu de dire `x.takeButton(this)`, l'objet Chien appelle `x.takeButton(new monBoutonInterne())`.

JTLP : Est-ce que je peux avoir un exemple clair ?

Objet interne : Vous souvenez-vous du panneau que nous avons utilisé quand nous avons créé notre propre sous-classe de JPanel ? Cette classe est une classe séparée, non une classe interne. Et c'est bien, parce qu'elle n'a pas besoin d'un accès spécial aux variables d'instance de l'IHM principale. Mais si elle en avait besoin ? Si nous créons une animation, et qu'elle obtenait ses coordonnées de l'application principale (par exemple en fonction de quelque chose que l'utilisateur fait quelque part ailleurs dans l'IHM). Dans ce cas, si nous faisons du panneau une classe interne, il devient une sous-classe de JPanel, tandis que la classe externe est libre d'être une sous-classe d'autre chose.

JTLP : Je vois ! Et de toute façon le panneau n'est pas suffisamment réutilisable pour être une classe séparée, puisque ce que nous dessinons est spécifique à cette seule application.

Objet interne : Voilà ! Vous y êtes !

JTLP : Bien. Nous pouvons donc passer à la nature de la relation que vous entretenez avec l'instance externe.

Objet interne : Mais qu'est-ce que vous avez tous ? Il n'y a donc pas assez de ragots sordides sur le polymorphisme ?

JTLP : Vous n'imaginez pas ce que le public est prêt à payer pour un bon scandale. Donc quelqu'un vous crée et vous vous trouvez instantanément lié à l'objet externe ? Non ?

Objet interne : Oui. Certains parlent à ce propos de mariage de raison. Nous n'avons pas notre mot à dire sur l'objet auquel nous sommes liés.

JTLP : Pour reprendre l'analogie, pouvez-vous divorcer et vous remarier avec quelqu'un d'autre ?

Objet interne : Non, c'est pour la vie entière.

JTLP : La vie de qui ? La vôtre ? Celle de l'objet externe ? Les deux ?

Objet interne : La mienne. Je ne peux être lié à aucun autre objet externe. Ma seule issue est le ramasse-miettes.

JTLP : Et l'objet externe ? Peut-il être associé à d'autres objets internes ?

Objet interne : Nous y sommes. C'est ça que vous vouliez ? Oui, mon prétendu « partenaire » peut en avoir autant qu'il en a envie.

JTLP : Est-ce de la... monogamie en série ? Ou est-ce qu'il peut les avoir tous en même temps ?

Objet interne : Tous en même temps. Vous êtes content ?

JTLP : Logique. Mais n'oubliez pas que c'est vous qui chantiez les louanges de « l'implémentation multiple d'une même interface ». Et si la classe externe a trois boutons, il est normal qu'elle ait trois classes internes (et donc trois objets) pour gérer les événements.

Merci pour tout. Tenez, voilà un mouchoir.

20 Graphique et événementiel

Qui suis-je ?



Un groupe de composants Java joue à « Qui suis-je ? ». Ils vous donnent des indices et vous devez essayer de deviner qui ils sont. On suppose qu'ils disent toujours la vérité. S'ils disent quelque chose qui peut être vrai de plusieurs d'entre eux, choisissez tous ceux auxquels l'assertion peut s'appliquer.

Les participants de ce soir : `setSize()`, objet interne, `addActionListener()`, objet event, `javax.swing`, `Graphics2D`, `repaint()`, source d'événement, `paintComponent()`, composant swing, `actionPerformed()`, interface écouteur, `JFrame`, événement.

1. Je tiens l'IHM dans mes mains.
2. Tout type d'événement en a une.
3. La méthode clé de l'écouteur d'actions.
4. Définit la taille du `JFrame`.
5. Vous lui ajoutez du code sans jamais l'appeler.
6. Quand l'utilisateur fait quelque chose, c'est un _____.
7. La plupart sont des sources d'événements.
8. Je renvoie des données à l'écouteur.
9. Une méthode `addXxxListener()` dit qu'un objet est une _____.
10. Comment un écouteur d'actions s'enregistre.
11. Méthode qui contient tout le code graphique d'un composant.
12. Je suis généralement lié à une autre instance.
13. Le "g" de (`Graphics g`) est en réalité de ce type.
14. La méthode qui (re)lance `paintComponent()`.
15. Le package où résident les composants Swing.

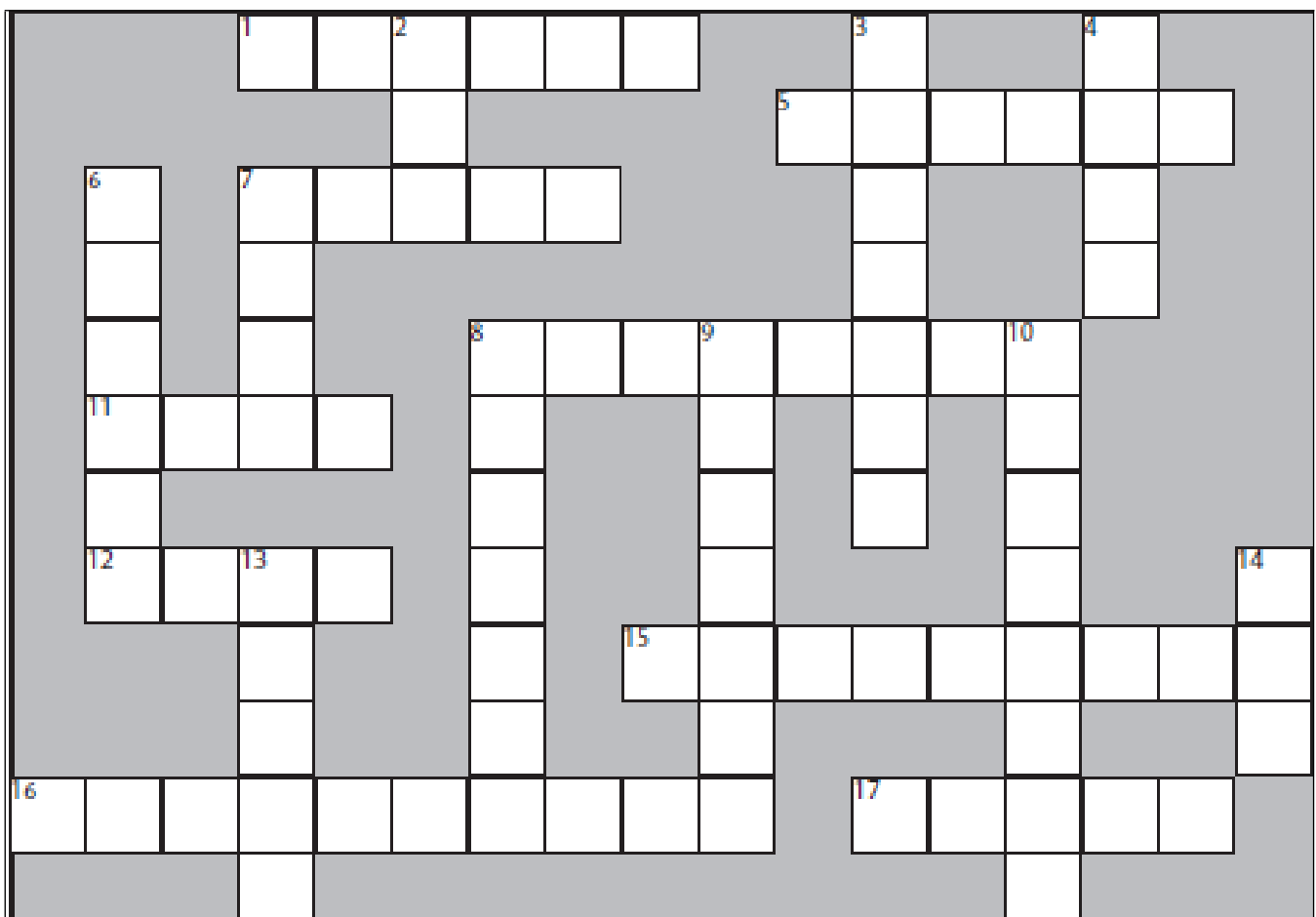
Mot croisé

Horizontalement

1. Sert à créer un panneau
5. Pour le milieu
7. Look Java
8. Pour créer des composants graphiques
11. Look Macintosh
12. Région orientale
15. Composant d'une boîte de dialogue
16. Ils se produisent
17. Région septentrionale

Verticalement

2. Abstract Windowing Toolkit
3. Fixe une taille
4. Région occidentale
6. Sert à créer un cadre
7. Donne le choix à l'utilisateur
8. Contraire de 3 vertical
9. Il en faut pour les textes
10. Mémorisera
13. Package indispensable pour les interfaces graphiques
14. X ou Y




21 Et pour conclure

CATASTROPHE OO !

Le jeu télévisé préféré d'Objectville

Pour éviter les risques	Concepteurs célèbres	Briques de code	Maintenance et réutilisation	Névroses logicielles
100 €	100 €	100 €	100 €	100 €
200 €	200 €	200 €	200 €	200 €
300 €	300 €	300 €	300 €	300 €
400 €	400 €	400 €	400 €	400 €

Bonjour et bienvenue à CATASTROPHE OO, le jeu télévisé préféré d'Objectville. Nous avons beaucoup de réponses OO ce soir, j'espère que vous êtes prêt à poser les **bonnes** questions.



CATASTROPHE OO !

Le jeu télévisé préféré d'Objectville

Pour éviter les risques	Concepteurs célèbres	Briques de code	Maintenance et réutilisation	Névroses logicielles
100 €	100 €	100 €	100 €	100 €
200 €	200 €		200 €	200 €
300 €	300 €			
400 €	400 €			

Cette brique de code a un double rôle : définir un comportement qui s'applique à plusieurs types et être le centre d'attention préféré des classes qui utilisent ces types.

« Qu'est-ce que _____ ? »

CATASTROPHE OO !

Le jeu télévisé préféré d'Objectville

Pour éviter les risques	Concepteurs célèbres	Briques de code	Maintenance et réutilisation	Névroses logicielles
100 €	100 €	100 €	100 €	100 €
200 €	200 €		200 €	200 €
300 €				300 €
400 €			400 €	400 €

Elle a permis d'éviter plus de problèmes de maintenance que n'importe quel autre principe OO ayant jamais existé. Grâce à elle, les modifications nécessaires pour que le comportement d'un objet varie restent locales.

« Qu'est-ce que _____ ? »

CATASTROPHE OO !

Le jeu télévisé préféré d'Objectville

Pour éviter les risques	Concepteurs célèbres	Briques de code	Maintenance et réutilisation	Névroses logicielles
100 €	100 €	100 €	100 €	100 €
200 €	200 €		200 €	200 €
300 €	300 €	300 €		300 €
	400 €			400 €

Toutes les classes devraient s'assurer de n'avoir qu'une seule raison de faire cela. C'est la cause de la mort de nombreux logiciels mal conçus.

« Qu'est-ce que _____ ? »

Vous vous en êtes bien sorti, mais il est temps de passer à la CATASTROPHE FINALE. Vous trouverez ci-dessous le diagramme de classe pour une application qui n'est pas très souple. Pour montrer que vous pouvez véritablement éviter une catastrophe OO, vous devez noter comment vous changeriez cette architecture. Vous aurez besoin de tous les principes dont nous avons parlé, alors prenez votre temps. Bonne chance !

Finale

CATASTROPHE !

