1

Exercice 1.2

Un référent ne peut désigner qu'un seul objet puisqu'il s'agit en effet de l'adresse de l'objet et qu'un objet ne peut être placé que dans une adresse unique.

Plusieurs référents peuvent ceci dit partager cette adresse et ainsi pointer vers le même objet. Il s'agit d'un système d'adressage indirect. Cette multiplication des référents est un aspect fondamental de la gestion mémoire par le « ramasse-miettes » dans les langages de programmation qui le permettent.

Un objet peut en référer un autre en, par exemple, pour une association forte entre deux objets, possédant parmi ses attributs l'adresse d'un autre objet.

En programmation orientée objet, tout ce que fait un objet est spécifié par sa classe. Cela permet notamment au compilateur de vérifier que le code est correctement écrit. C'est la classe qui définit également les parties communes à tous les objets : types d'attribut et méthodes.

Tout dépend de ce l'on appelle l'état d'un objet mais, en substance, toute modification des attributs d'un objet ne peut se faire que par l'exécution de méthodes qui sont prévues dans la définition de la classe de l'objet.

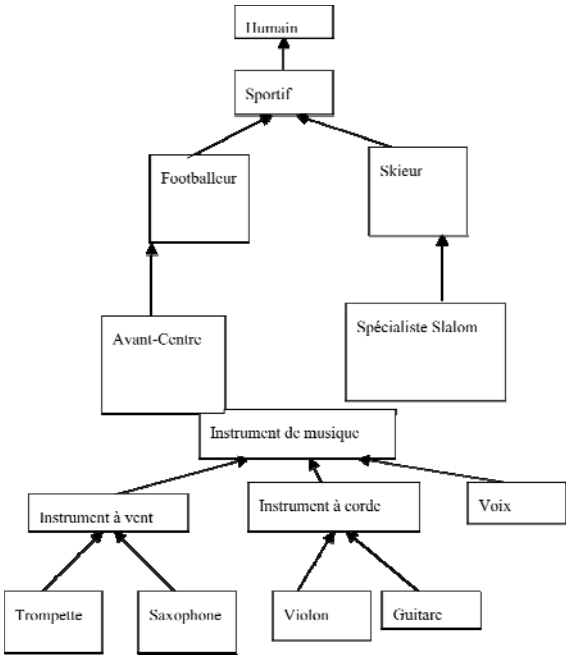
Cela signifie que la méthode « f(x) » s'exécute sur l'objet désigné par « a ». La classe de l'objet « a » contiendra en son sein la méthode « f(x) ».

La méthode « f(x) » soit être déclarée dans une des classes qui définit l'objet « a ». Il faut connaître le mécanisme d'héritage pour savoir que plus d'une classe peut être concernée, et que cette méthode doit se trouver dans une des classes ou superclasses qui décrit l'objet « a ».

On appelle « envoi de message » la présence dans le code d'une classe d'un appel de méthode s'exécutant sur un autre objet. Lors de l'exécution du code, un premier objet passera le relais à un deuxième pour l'exécution de cette méthode le concernant.

Il suffit que le message qui provoque l'exécution de la méthode sur le deuxième ait spécifiquement dans ses attributions la modification des attributs de ce deuxième. Rappelons-nous que le respect de l'encapsulation rend cette modification directe impossible pour le premier objet qui ne peut avoir un accès direct aux attributs du deuxième.

Exercice 1.3



3404lsqj

Exercice 1.5

Expéditeur	Destinataire
Chauffeur	Voiture
Footballeur	Ballon
Guitariste	Guitare
Télécommande	Télévision

2

Exercice 2.1

```
class Voiture
{
private int vitesseChange;
Voiture(int vitesseChange) {
this.vitesseChange = vitesseChange;
}
public void changeVitesseHaut(){
vitesseChange ++;
}
public void changeVitesseBas(){
vitesseChange --;
}
public void changeVitesseHaut(int deltaV) {
vitesseChange += deltaV;
}
}
public class Principale
{
public static void main(String[] args) {
Voiture v = new Voiture(0);
v.changeVitesseHaut();
}
}
```

Exercice 2.2

```
public void test() _ admis
```

```
public void test(double a) _ admis
```

```
public void test(int a, int b) _ admis
```

```
public int test(int a) _ Non admis car on modifie uniquement la valeur du retour et
on ne saurait quelle version chercher pour une simple instruction comme a.test(5).
```

Exercice 2.3

Le code Java ne compilera pas à cause simplement de la présence du `void` qui fait de cette méthode une autre méthode que le constructeur. Le compilateur ne trouve donc pas le constructeur lors de l'instruction `A unA = new A(5) ;`.

Le code C# ne compilera pas car il refusera que vous puissiez créer une méthode autre que le constructeur avec le même nom que la classe. En fait, il évite le genre d'erreur plus difficile à déceler dont le code Java pourrait être responsable. Seul le constructeur peut porter le nom de la classe.

Exercice 2.4

```
class Voiture
{
private int vitesseChange;
Voiture () {
vitesseChange = 0;
}
Voiture(int vitesseChange) {
this.vitesseChange = vitesseChange;
}
}
public class Principale
{
public static void main(String[] args) {
Voiture v = new Voiture(0);
}
}
```

Exercice 2.5

Seraient statiques la vitesse maximale (la même pour tous les véhicules), le nombre de vitesses (le même pour tous les véhicules), la capacité du réservoir (le même pour tous les véhicules), la puissance, le nombre de portières.

Exercice 2.6

Dans le code Java, dès qu'il y a création d'un constructeur, celui-ci écrase d'office le constructeur par défaut qui est sans argument. Vous ne pouvez donc plus créer un code en faisant un appel explicite à ce constructeur là car il n'existe plus.

Dans le code C#, le deuxième constructeur s'attend à recevoir deux entiers et non pas un double. Vous ne pouvez pas caster implicitement un double dans un entier. L'inverse est possible.

Dans le code C++, comme pour le code Java, en l'absence de constructeur par défaut sans argument (écrasé par les deux autres), la troisième instruction ne compilera pas.

Exercice 2.7

Dans le code Java, L'attribut `c` est déclaré non statique. Dès lors, il ne peut être appelé dans une méthode déclarée, elle, comme statique. Toute méthode statique ne peut faire appel qu'à des attributs et des méthodes déclarées également statiques.

Dans le code C#, et au contraire de Java, une méthode statique ne peut être appelée sur autre chose que la classe elle-même. Appelée sur un objet, le compilateur s'offusque, ce qui est somme toute assez logique.

Dans le code C++, la seule manière d'initialiser l'attribut statique `c` est en dehors de la classe `Test` par une instruction :

```
int Test::c = 5;
```

Exercices 2.8 et 2.9

```
class Point {
private int x,y,z;
Point (int x) {
this.x = x;
this.y = 0;
this.z = 0;
}
Point (int x, int y) {
this.x = x;
this.y = y;
this.z = 0;
}
Point (int x, int y, int z) {
this.x = x;
this.y = y;
this.z = z;
}
public void translate (int dx) {
x += dx;
}
public void translate (int dx, int dy) {
x += dx;
y += dy;
}
public void translate (int dx, int dy, int dz) {
x += dx;
y += dy;
z += dz;
}
}
public class TestPoint {
public static void Main() {
Point p1 = new Point (5);
Point p2 = new Point (6, 7, 8);
p1.translate(5);
p2.translate(7,8);
}
}
```

C'est la même chose pour le code C#.

4

Exercice 4.1

```
class Interrupteur {
private int position;
private Lampe l;
Interrupteur (Lampe l) {
position = 0;
this.l = l;
}
public void allume () {
position = 1;
l.allume();
}
public void eteint () {
position = 0;
l.eteint();
}
}
class Lampe {
private boolean allume;
Lampe () {
allume = false;
}
public void allume () {
allume = true;
}
public void eteint () {
allume = false;
}
}
public class PrincipalLampe {
public static void main(String[] args) {
Lampe l = new Lampe();
Interrupteur i = new Interrupteur(l);
i.allume();
}
}
```

Exercice 4.2

La relation de dépendance crée un lien temporaire entre les deux classes qui n'a d'existence que le temps d'un envoi de message alors que l'association fait de ce lien un véritable attribut de classe. On privilégiera la dépendance lorsque l'envoi de message ne doit se faire que pour une ou deux méthodes de l'objet expéditeur et n'a pas lieu d'être durant toute la durée de vie de ces objets expéditeurs.

Exercice 4.3

```
class ExpediteurAssociation {
Destinataire d;
ExpediteurAssociation (Destinataire d) {
this.d = d;
}
public void envoi () {
d.message();
}
}
class ExpediteurDependance {
ExpediteurDependance () {
}
public void envoi (Destinataire d) {
d.message();
}
}
class Destinataire {
public void message() {}
}
```

Exercice 4.4

```
class Expediteur {
private Destinataire1 d1;
Expediteur (Destinataire1 d1) {
this.d1 = d1;
}
public void envoi () {
System.out.println("expediteur
envoie");
d1.envoi1();
}
}
class Destinataire1 {
private Destinataire2 d2;
Destinataire1 (Destinataire2
d2) {
this.d2 = d2;
}
public void envoi1 () {
System.out.println("destinateur
1
envoie");
d2.envoi2();
}
}
class Destinataire2 {
Destinataire2 () {
}
public void envoi2 () {
System.out.println("destinateur
2
reçoit");
}
}
public class Test3Messages {
public static void
main(String[] args) {
Expediteur e = new Expediteur
(new
Destinataire1 (new
Destinataire2()));
e.envoi();
}
}
```

Exercice 4.5

```
import java.awt.*;
class Fenetre extends Frame {
Fenetre () {
}
public void jeMeFerme () {
// Code pour fermeture;
}
}
class Souris {
public void click (Fenetre f) {
// Selon l'endroit du click
f.jeMeferme();
}
}
```

5

Exercice 5.3

```
class AutoMessage {
AutoMessage am;
AutoMessage () {}
AutoMessage (AutoMessage am) {
this.am = am;
}
public void message() {
System.out.println("ceci est un
message");
}
public void jeMeParle () {
message();
}
public void jeParleAUnCollegue() {
am.message();
}
}
public class TestAutoMessage {
public static void main(String[] args) {
AutoMessage am = new AutoMessage();
AutoMessage am2 = new
AutoMessage(am);
am.jeMeParle();
am2.jeParleAUnCollegue();
}
}
```

Exercice 5.4

En Java, la compilation est dynamique et donc le code de la classe `B` se compilera automatiquement dès compilation du code de la classe `A`. Il ne faut procéder à aucune liaison explicite. Cela est rendu possible par la dénomination des classes semblable à la définition des fichiers qui les contiennent. Lors de la compilation du fichier A.java, et à la découverte de l'association vers la classe B, le compilateur partira à la recherche du fichier B.java (à l'aide des « imports », s'il ne se trouve pas dans le même package) et le compilera s'il a été changé depuis la dernière compilation.

En C# et C++, il faut lier explicitement les fichiers qui contiennent les classes entrant dans un lien d'association ou de dépendance.

Exercice 5.5

Le nom complet sera : `as.as1.A`.

6

Exercice 6.1

Le code Java affichera :

```
i = 6
i = 5
```

Le code C# affichera :

```
i = 6
i = 5
i = 6
i = 6
```

Exercice 6.2

Le code C++ affichera :

```
i = 6
i = 5
i = 6
i = 6
```

Exercice 6.3

Le code Java affiche :

```
i = 20

Si nous remplaçons j par k, le
résultat devient i=25
```

Exercice 6.4

Le code C++ affiche :

```
i = 6
i = 5
i = 10
i = 5
i = 10
```

Exercice 6.5

Le code Java affiche :

```
i= 5
```

Exercice 6.6

Tant en C# qu'en Java, les objets sont toujours passés par référent, car c'est leur adresse qui est dupliquée dans la méthode (et donc on se retrouve à travailler avec l'objet original). En C++, les objets comme tout argument sont passés par défaut par valeur avec la problématique d'un clonage intempestif des objets durant l'exécution des codes C++. C'est la raison première de l'existence en C++ du constructeur par copie, qui se met en action dès qu'un objet est passé par argument. Java et C# ont opté pour la solution plus logique, consistant à passer l'objet original et non une copie de celui-ci.

C#, par ailleurs, permet par les extensions `ref` et `out`, le passage également d'argument au type prédéfini (`int`, `double`…) par référent alors qu'ils le sont par défaut par valeur.

7

Exercice 7.1

```
public class Calendrier {
private int date;
Calendrier (int date) {
this.date = date;
}
public String lireDate() {
return "" + date;
}
public static void main(String[] args) {
Calendrier c = new Calendrier
(12052007);
System.out.println(c.lireDate());
}
}
```

Exercice 7.2

20, deux méthodes d'accès : `get` et `set` par attribut. Sachez néanmoins que même ces deux méthodes se devraient d'être rarement sollicitées, car les attributs d'une classe sont souvent exploités dans des actions autres que juste la lecture ou l'écriture.

Exercices 7.3 et 7.4

```
public class CompteEnBanque {
private double solde;
CompteEnBanque () {
solde = 0;
}
public void credit (double montant) {
if (solde + montant > 0) {
solde += montant;
}
}
public void debit (double montant) {
if (solde - montant > 0) {
solde -= montant;
}
}
public int lireSolde () {
return (int)solde;
}
public static void main(String[] args) {
CompteEnBanque c = new
CompteEnBanque();
c.credit (100);
c.debit(200);
c.debit(50);
System.out.println(c.lireSolde());
}
}
```

8

Exercice 8.2

Feront partie de l'interface les méthodes : `tourne`, `accélère` et `changeVitesse`.

Exercice 8.3

Une méthode « private » sera accessible pour peu qu'elle soit appelée par une méthode publique de cette même classe. Cette autre classe en faisant appel dans son code à la méthode publique déclenchera indirectement l'exécution de la méthode privée.

Exercice 8.4

L'assemblage peut assurer une forme d'encapsulation dans la mesure où il est possible de rendre accessible méthode et attribut d'une classe à l'autre, pour autant que ces classes soient définies dans un même assemblage. L'encapsulation peut donc se restreindre au seul assemblage, comme si toutes les classes de l'assemblage étaient amies entre elles.

Exercice 8.5

C'est une forme suprême d'encapsulation car la classe implémentant les services décrits dans l'interface peut modifier cette implémentation sans affecter d'aucune sorte les classes qui y font appel. L'interface prémunit les classes bénéficiaires de cette interface des changements survenus dans l'implémentation des codes des services prévus par l'interface. L'interface est souvent considérée comme le mode d'emploi de la classe et donc de tous les objets issus de cette classe.

Exercice 8.6

Car si elles l'étaient, l'impact d'une modification dans une des classes amies aurait des répercutions loin dans le code, et il faut se rappeler que la principale raison d'être de l'encapsulation est d'éviter l'impact de modification des codes lors du développement de ceux-ci.

9

Exercice 9.2

Lors de l'effacement de l'objet O3, celui-ci pointant vers un objet O2, lui-même pointant vers un objet O1, il restera 2 objets en mémoire perdus et inutilisables jusqu'à l'interruption du code.

Exercice 9.3

Il ne devrait plus rester aucun objet en mémoire après l'appel du ramasse-miettes, vu que les objets disparaissent avec la disparition de leurs référents. Chaque objet n'étant pointé que par un référent, la suppression du premier O3 entraînera dans sa suite la suppression des deux autres.

Exercice 9.4

Ce code donnera un nombre invraisemblable à l'écran du fait que l'on a effacé l'objet sur lequel s'effectue la deuxième méthode. On se retrouve typiquement dans la situation dite du « pointeur fou ». Pour bien faire, il faudrait annuler le pointeur après l'effacement de l'objet vers lequel il pointe de façon à le rendre inutilisable.

Exercice 9.5

Un simple effacement d'objets basé sur le seul comptage des référents est ici rendu difficile par la présence d'un cycle de référence. Le « ramasse-miettes » doit détecter la présence de tel cycle dans lequel, bien que chaque objet possède un référent vers un autre, l'ensemble du cycle s'avère inaccessible et devrait être supprimé dans sa globalité par le ramasse-miettes.

Exercice 9.6

Le nombre 10.

Exercice 9.8

C# a repris l'idée du ramasse-miettes de Java mais autorise également une gestion mémoire des objets pas mémoire pile (gestion dite aussi par valeur), dès lors que l'on crée ces objets comme instance de « structure » et non plus de classe.

10

Exercices 10.1

L'exercice 10.1-4 est traité dans le détail au chapitre 19 du livre.

Exercice 10.2

```
class A {
private :
int x1 ;
E* e ;
public :
void faireX() {
e->faireE() ;
}
} ;
class B : public A {
private :
int y1 ;
public :
void faireY() {}
} ;
class C : public A {
public :
void faireZ() {}
};
class D : public A {
private :
int z1 ;
int z2 ;
public :
void faireF() {}
} ;
class E {
private :
double x1 ;
public :
faireE() {}
}
```

Exercice 10.3

```
class A {
private int unX ;
private int unY ;
public void faireX() {}
public void faireY() {}
public void faireZ() {}
}
class B extends A {
private int unC ;
private int unD ;
private ArrayList<E> lesE ;
private D unD ;
public void faireC() {}
public void faireD() {}
}
class E extends A {
private double unI ;
```

```
private string unJ ;
private B unB ;
private D unD ;
public void faireC() {}
public void faireD() {}
}
class D extends B implements interface-C {
private double unI ;
private string unJ ;
private B unB ;
private E unE ;
private ArrayList<D> B ;
public void faireC() {}
public void faireD() {}
}
```

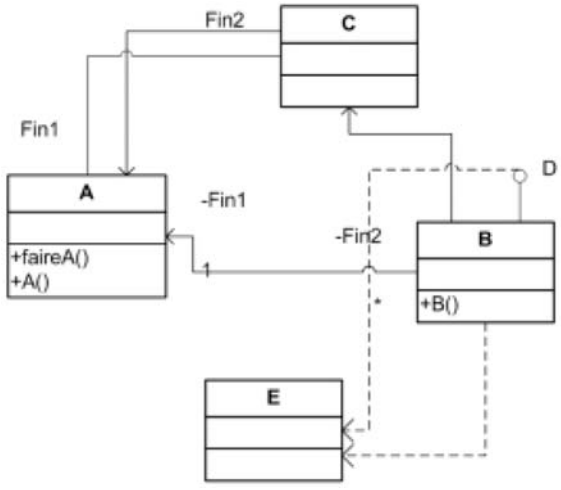
Exercice 10.4

```
class FigureGeometrique {
private int positionX ;
private int positionY ;
public void dessine() {}
public void changeLaTaille() {}
}
class CercleDansCarre : FigureGeometrique {
private Rectangle r ;
private Cercle c ;
CercleDansCarre () {
r = new Rectangle() ;
c = new Cercle() ;
}
public void changeLaTaille() {
c.getLargeur() ;
r.setNouvelleLargeur(nL) ;
r.setNouvelleHauteur(nH) ;
c.setNouveauRayon(nR) ;
}
}
class Cercle : FigureGeometrique {
private int rayon ;
private CercleDansCarre c ;
public int getRayon() {}
public void setNouveauRayon() {}
}
class Rectangle: FigureGeometrique {
private int largeur ;
private int hauteur ;
private CercleDansCarre c ;
public int getLargeur () {}
public void setNouvelleLargeur(int l) {}
public void setNouvelleHauteur(int h) {}
}
class Utilisateur {
private CercleDansCarre cc ;
public static void main(String[] args) {
cc.changeLaTaille();
cc.dessine();
}
}
```

Exercice 10.5

```
class ElementDeMatiere {
} ;
class Atome :public ElementDeMatiere {
private :
Molecule m ;
} ;
class Molecule :public ElementDatiere {
private :
Atome* lesAtomes ;
public :
Molecule () {
lesAtomes = new Atome[] ;
}
} ;
class Cellule :public Molecule {
private :
Molecule* lesMolecules ;
public :
Cellule () {
lesMolecules = new Molecule[] ;
}
} ;
Class ADN :public Cellule {
private :
ADN* unADN ;
public :
void combine() {
unADN->combine() ;
unADN->cree() ;
}
void cree() {}
} ;
```

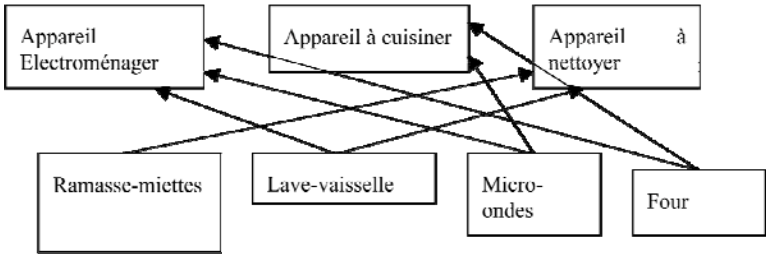
Exercice 10. 6



43041sqj

11

Exercice 11.1



44041sqj

Exercice 11.3

En Java ce sera :

```
class D extends B {}
class B extends A {}
class C extends A {}
class A {}
```

Exercice 11.4

En C++, ce sera :

```
class D : public B, public C {}
class B : public A {}
class C : public A {}
```

En Java et C#, il faudra remplacer `A`, et `B` ou `C` par des interfaces car le multihéritage est impossible.

```
interface A {}
interface B extends A {}
class C implements A {}
class D extends C implements B
```

Exercice 11.5

En Java, pas de multihéritage.

En C#, la classe `Exo2` doit impérativement implémenter la méthode abstraite `jeTravaillePourInterface()`.

En C++, il faut préciser quelle version de la méthode `jeTravaillePourLaClasse()` on veut exécuter, par exemple : `O1 ::jeTravaillePourLaClasse()`.

Exercice 11.6

En Java, l'instruction : `unO2 = unO1` est impossible, le casting implicite est possible d'une sous-classe dans sa super-classe, mais pas l'inverse à cause du principe de substitution.

En C#, même problème pour la dernière instruction dans laquelle on cherche à placer un objet typé super-classe dans un objet typé sous-classe et ce type de casting implicite est interdit par le compilateur.

Même problème en C++.

Exercice 11.7

Car les sous-classes resteront affectées par toute modification des méthodes ou attributs « protected » de la super-classe, ce qui va à l'encontre de l'encapsulation.

Exercice 11.8

Un héritage « private », transforme tout ce qui est public dans la super-classe en privé pour ses sous-classes (et donc inaccessible pour les sous-classes de celles-ci). C'est un degré d'encapsulation supplémentaire, un peu ardu à maîtriser, et que ni Java ni C#, dans un souci de simplicité, n'a trouvé nécessaire de maintenir.

Exercice 11.9

Par le principe de substitution, c'est la première assertion qui est correcte. Une Ferrari est une voiture pas l'inverse.

Exercice 11.10

On caste toujours dans la sous-classe. C'est donc la première instruction qui est correcte. La deuxième forme de casting est toujours implicite et donc inutile.

Exercice 11.11

Car il y a des multitudes de versions de celles-ci, en rajoutant au fur et à mesure de nouvelles spécifications.

12

Exercice 12.1

Diagramme de classe :

Une esquisse de code serait :

```
import java.util.ArrayList;


public class Banque {
	private ArrayList<CompteBanque> lesComptes;

	public Banque() {
		lesComptes = new ArrayList<CompteBanque>();
	}

	public void addCompte() {
		lesComptes.add(new CompteCourant());
	}

}

abstract class CompteBanque {
	private int solde;
	abstract public void retrait(int montant);
	abstract public void depot(int montant);
	abstract public void calculInteret();
}

class CompteCourant extends CompteBanque {
	public void retrait(int montant) {}
	public void depot(int montant){}
	public void calculInteret() {}
}

class LivretEpargne extends CompteBanque {
	public void retrait(int montant) {}
	public void depot(int montant){}
	public void calculInteret() {}
}
```

Exercice 12.3

Le code Java affichera :

```
LePon a fait 3 voix
Laguillerette a fait 2 voix
StChasse a fait 0 voix
LeChe a fait 1 voix
Madeleine a fait 0 voix
SuperLier a fait 0 voix
Jaudepis a fait 2 voix
Tamere a fait 0 voix
```

Exercice 12.4

Le code Java affichera `hello` + l'adresse de l'objet en question.

Exercice 12.5

```
class A {
public void a() {
System.out.println("a de A");
}
public void b() {
System.out.println("b de A");
}
}
class B extends A {
public void b() {
System.out.println("b de B");
}
public void c() {
System.out.println("c de B");
}
}
public class Correction2 {
public static void main(String[] args) {
A a1=new A();
A b1=new B();
B a2=new A(); // NE PASSE PAS A LA COMPILATION
B b2=new B();
a1.a();
b1.a();
a2.a();
b2.a();
a1.b();
b1.b();
a2.b();
b2.b();
a1.c(); // NE PASSE PAS A LA COMPILATION
b1.c(); // NE PASSE PAS A LA COMPILATION
a2.c();
b2.c();
((B)a1).c(); // CETTE INSTRUCTION PROVOQUE A L'EXECUTION UN CLASS CAST EXCEPTION;
((B)b1).c();
((B)a2).c();
((B)b2).c();
}
}
```

Exercice 12.6

Le code C++ affichera :

```
Je fais un gros
rugissement
Je fais un petit
barrissement
Je fais un gros
ronflement
Je mange beaucoup de
choucroute
Je mange beaucoup de
choucroute
Je mange beaucoup de
radis beurre
```

Exercice 12.7

Le code C# affichera :

```
Attention violon desaccorde
Attention pas d'accordeur de piano
Tout va bien
on teste
```

13

Exercice 13.1

Car on ne serait que faire lorsque la méthode abstraite est exécutée sur un objet instance de cette classe. Il est donc tout à fait logique d'interdire la création d'objet à partir d'une classe abstraite.

Exercice 13.2

Pour la même raison que précédemment. Que faire si l'on déclenche la méthode abstraire sur un objet issu de la sous-classe ?

Exercice 13.3

C++ considère que dès lors qu'une méthode est définie abstraite (virtuelle pure) dans une classe, automatiquement celle-ci en devient abstraite. C++ n'autorise pas une classe à devenir abstraite sans qu'au moins une méthode le soit.

Exercice 13.4

Car en étant abstract, elle est d'office « virtual » en ceci qu'elle exige d'être redéfinie.

Exercice 13.5

```
abstract class MoyenDeTransport {
public abstract void consomme();
}
class Voiture extends MoyenDeTransport {
public void consomme () {
System.out.println("a la maniere
voiture");
}
}
class Moto extends MoyenDeTransport {
public void consomme () {
System.out.println("a la maniere
moto");
}
}
class Camion extends MoyenDeTransport {
public void consomme () {
System.out.println("a la maniere
camion");
}
}
```

Exercice 13.7

Le code Java affiche :

```
C'est super
C'est super
vivement le bar
vivement la bibliothe
A vos ordres chef
A vos ordres chef
A vos ordres chef
Salut les cocos
et merde
A vos ordres chef
Salut les cocos
et merde
A vos ordres chef
```

Exercice 13.8

Le code C# affiche :

```
je fais un petit
groin groin
et je m'endors en fermant les
yeux
je fais un petit
cot cot
et je m'endors sur mon perchoir
je fais un gros
chuuuuut
et je m'endors a côte de ma
vache
```

Exercice 13.9

Le code Java affiche :

```
le bananier dit:
lumosite parfaite
plus d'eau
l'olivier dit:
lumosite parfaite
plus d'eau
le magnolia dit:
je suis aveuglee
plus d'eau
le bananier dit:
lumosite parfaite
blou bloup
l'olivier dit:
lumosite parfaite
blou bloup
le magnolia dit:
je crame!!
blou bloup
le bananier dit:
more light please
plus d'eau
l'olivier dit:
more light please
plus d'eau
le magnolia dit:
more light please
plus d'eau
```

Exercice 13.11

Il faut déclarer les attributs de la classe `A` « protected » ou prévoir des méthodes d'accès. Il faut redéfinir la méthode `decrisToi()` dans la classe `B` vu qu'elle est abstraite dans la classe `A`. Il faut faire appel au constructeur de la classe `A` pour l'initialisation des attributs hérités de `A` : `super (a,b,c) ;`.

14

Exercice 14.1 et 14.2

En C#, cela donnerait :

```
class CompteEnBanque
{
private int solde;
public CompteEnBanque(int solde)
{
this.solde = solde;
}
public static bool operator
==(CompteEnBanque compte1, CompteEnBanque compte2)
{
if (compte1.solde == compte2.solde)
{
return true;
}
else
{
return false;
}
}
public static bool
operator !=(CompteEnBanque compte1, CompteEnBanque
compte2)
{
return !(compte1 == compte2);
}
public static CompteEnBanque operator
+(CompteEnBanque c1, CompteEnBanque c2)
{
return new CompteEnBanque(c1.solde +
c2.solde);
}
}
```

En C++, ce serait très semblable.

Exercice 14.3

Pour nous forcer à la redéfinir dans toutes les classes (qui sont toutes par définition sous-classe de la classe `Object`). La méthode `equals()` doit être redéfinie car, par défaut, elle ne porte que sur les référents, c'est-à-dire les adresses des objets (comme le `==`), il importe donc de la rendre plus générale et de la faire porter sur les attributs de différents objets. Étant « protected », elle ne peut être utilisée en dehors de la sous-classe et il faut donc qu'elle soit automatiquement redéfinie dans la sous-classe comme « public » cette fois.

Exercice 14.4

Car celle-ci, dès le moment où elle s'applique sur deux objets passés en argument, ne peut s'appeler qu'à partir de la classe et non plus à partir d'un objet.

Exercice 14.5

Car l'appel de la méthode `Equals()` renvoie bien à la notion d'envoi de message.

Exercice 14.6

```
class CompteEnBanque
{
private int solde;
private string titulaire;
private ArrayList lesEmprunts;
public CompteEnBanque(int solde, string
titulaire)
{
this.solde = solde;
this.titulaire = titulaire;
lesEmprunts = new ArrayList();
}
public int Solde
{
get
{
return solde;
}
set
{
solde = value;
}
}
public void calculInteret()
{
Console.WriteLine("interet par défaut");
}
public void addEmprunt (Emprunt e) {
lesEmprunts.Add(e);
}
public static bool operator
==(CompteEnBanque compte1, CompteEnBanque compte2)
{
int Somme1 = 0;
int Somme2 = 0;
foreach (Emprunt e in
compte1.lesEmprunts)
{
Somme1 += e.Montant;
}
foreach (Emprunt e in
compte2.lesEmprunts)
{
Somme2 += e.Montant;
}
if (Somme1 == Somme2)
{
return true;
}
else
{
return false;
}
}
public static bool
operator !=(CompteEnBanque compte1, CompteEnBanque
compte2)
{
return !(compte1 == compte2);
}
}
```

15

Exercice 15.1

En Java, cela donnerait :

```
interface IA {}
interface IB {}
interface IC extends IA,IB {}
interface ID extends IA,IB {}
class E implements IC,ID {}
```

En C# :

```
interface IA {}
interface IB {}
interface IC : IA,IB {}
interface ID : IA,IB {}
class E : IC,ID {}
```

Exercice 15.2

La flèche d'interface doit aller de la classe `B` à l'interface `IA`. La classe n'a pas besoin d'interagir directement avec l'implémentation de la classe. C'est la pratique de l'encapsulation poussée à son paroxysme.

Exercice 15.3

Comme il n'y a pas lieu d'y mettre des attributs et que les méthodes n'ont pas de corps d'instruction dans les interfaces, toutes les ambiguïtés possibles en cas de duplication des mêmes dénominations dans des interfaces distinctes sont levées.

Exercice 15.5

Une méthode ne peut être déclarée comme « private » dans une interface ? Toutes les méthodes doivent être implémentées. Il manque ici la méthode `faire3A()`.

Exercice 15.6

Il ne peut y avoir de « public » devant la déclaration des interfaces. Dans la déclaration de l'héritage, la classe (la seule classe !!) doit toujours venir avant l'interface.

Exercice 15.7

Si la classe « héritière » n'a pas, en son sein, redéfinit toutes les méthodes. Elle ne constituera dès lors pas le dernier niveau d'héritage.

Exercice 15.8

Car le fichier interface est juste la signature des méthodes sans leur implémentation. Toutes les méthodes rajoutées dans le fichier implémentation auront un statut différent de celles prédéfinies dans l'interface.

Exercice 15.9

Oui, c'est même une excellente pratique que les classes ne se dévoilent entres elles que leur interface et qu'elles se dissimulent leur implémentation.

16

Exercice 16.1

Car la classe A codée dans l'ordinateur A n'a nul besoin de connaître l'implémentation de la classe B codée dans l'ordinateur B avec laquelle la première doit communiquer. Seule la connaissance par A de l'interface de B permet la communication.

Exercice 16.2

IDL étant très très proche de C++, la traduction de l'IDL se fait très naturellement et simplement en C++.

Exercice 16.3

RMI permet le polymorphisme et donc l'association dynamique (pendant l'exécution), car le code des classes peut s'échanger entre les machines pendant l'exécution. Cet échange de code n'est pas possible par CORBA qui doit connaître tout à la compilation.

Exercice 16.4

XML résulte du besoin de séparer dans la réalisation de pages Web, la présentation de ces pages de leur contenu informationnel. XML est ce langage de description de contenu. Il était donc assez naturel de reprendre ce langage de description pour les méthodes rendues disponibles sur le Web.

Exercices 16.7 et 16.8

Cela est rendu possible par l'existence de proxy tant du côté serveur que du côté client qui rendent transparent toute la couche transport et permet au programmeur de ne se préoccuper que de la couche logique, de la raison qu'ont ses classes de communiquer plutôt que de la manière matérielle dont elles vont le faire.

Exercice 16.9

Jini permet la découverte des proxys en ligne. Il n'oblige pas le programmeur de connaître au départ les services qu'il va utiliser. Il peut les découvrir lors de l'exécution du code.

Exercices 16.10 et 16.11 et 16.12

Les objets doivent se connaître par leur nom pour pouvoir se solliciter mutuellement. C'est en effet le cas de RMI et CORBA. Pour les services Web, seul le nom de la méthode suffit, car la communication est sans mémoire et aucun objet n'est nécessaire du côté du serveur pour maintenir les résultats des interactions advenues jusqu'présent.

Les services Web compensent cela en créant des variables de session du côté serveur qui maintiennent les informations sur les interactions.

Oui, c'est même une excellente pratique que les classes ne se dévoilent entres elles que leur interface et qu'elles se dissimulent leur implémentation.

17

Exercice 17.1

Il produira par exemple :

```
103 Thread-0
104 Thread-1
105 Thread-1
106 Thread-1
107 Thread-1
108 Thread-1
109 Thread-1
110 Thread-1
111 Thread-1
112 Thread-1
113 Thread-1
114 Thread-1
115 Thread-1
116 Thread-1
117 Thread-1
118 Thread-1
119 Thread-1
120 Thread-0
121 Thread-0
122 Thread-0
123 Thread-0
124 Thread-0
125 Thread-0
126 Thread-0
127 Thread-0
128 Thread-0
129 Thread-0
130 Thread-0
131 Thread-0
132 Thread-0
133 Thread-0
134 Thread-0
135 Thread-0
136 Thread-0
137 Thread-1
138 Thread-1
139 Thread-1
140 Thread-1
141 Thread-1
142 Thread-1
143 Thread-1
144 Thread-1
145 Thread-1
146 Thread-1
147 Thread-1
148 Thread-1
149 Thread-1
150 Thread-1
151 Thread-1
152 Thread-1
153 Thread-1
154 Thread-0
155 Thread-0
```

Pour avoir cette alternance, il suffit de mettre un `sleep(10)` à l'intérieur de la boucle et ceci dans un `try – catch`.

Exercice 17.2

Il suffit de synchroniser la méthode imprime.

```
public synchronized void
  imprime(String[] s) {
```

Exercice 17.5

Pour que les sessions transactionnelles des clients puissent être traitées simultanément

Exercice 17.6

Car elle ne concerne toujours que le seul thread en cours, elle n'a donc pas lieu de d'exécuter sur des objets divers. Elle ne s'exécute toujours qu'à même un seul objet.

Exercice 17.7

Bien évidemment, de manière à ce que cet accès soit cohérent et qu'un accès concurrent ne vienne pas interrompre un précédent qui débuterait les changements, mais sans avoir fait toutes les mises à jour que ces changements entraînent, laissant la base de données dans un état incorrect.

Exercice 17.8

Car c'est le système d'exploitation qui s'occupe de l'allocation des durées d'exécution des différents threads. Et cette durée dépend également en partie de la situation de tous les autres threads dont doit s'occuper le système d'exploitation, y compris des siens propres.

18

Exercices 18.1, 18.2

Dans la communication de type « événement », l'expéditeur n'a pas à connaître explicitement ses destinataires. Ces derniers peuvent simplement s'inscrire de façon à être informé des modifications d'état d'un objet qui est susceptible de les intéresser. Lorsque cette modification se produit et, quasiment à l'insu de l'expéditeur, les destinataires en seront informés et feront de cette information ce qui leur chante.

Exercice 18.3

En Java, cela donnerait :

```
import java.util.*;
public class Lotterie extends Observable {
private int numero;
public Lotterie() {
numero = 0;
}
public static void main(String[] args) {
Lotterie l = new Lotterie();
Joueur j = new Joueur(25);
l.addObserver(j);
l.demarre();
}
public int getNumero () {
return numero;
}
public void demarre () {
for (int i=0; i<50; i++) {
numero++;
System.out.println(i);
setChanged();
notifyObservers();
}
}
}
class Joueur implements Observer {
private int numero;
Joueur (int numero) {
this.numero = numero;
}
public void update (Observable o, Object arg)
{
if (((Lotterie)o).getNumero() ==
numero) {
System.out.println("Wouahhhhhh !!!! J'ai
gagné");
}
}
}
```

Exercice 18.4

En C#, cela donnerait :

```
using System;
using System.Threading;
```

```
public class Test {
public static void Main(String[] args) {
Reveil reveil = new Reveil();
Dormeur marcel = new
Dormeur("Marcel");
reveil.client += new trop
(marcel.seReveiller);
marcel.seCoucher(20);
Dormeur maurice = new
Dormeur("Maurice");
reveil.client += new
trop(maurice.seReveiller);
reveil.client += new
trop(maurice.seReveiller);
maurice.seCoucher(20);
reveil.demarre(100);
}
}
public delegate void trop();
class Reveil {
private int temps;
public trop client;
public void demarre(int temps) {
this.temps = temps;
Thread threadReveil = new Thread(new
ThreadStart(tictac));
threadReveil.Start();
}
public void tictac() {
for (int i=0; i < temps; i++) {
Console.Write("tic...tac...");
Thread.Sleep(200);
}
Console.WriteLine();
Console.WriteLine("Drrrrriiiiiiiiiiiiiiiiiii
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
iinnnnnnnnnnnnnnnnnnnnng !!!!");
client();
}
}
class Dormeur {
Thread threadDormeur;
String nom;
public Dormeur(string nom){
this.nom = nom;
}
public void seCoucher(int temps){
Console.WriteLine("Bonjour, je
m'appelle " + nom + " et je suis fatigué");
Console.WriteLine("Je vais dormir " +
temps + " minutes");
threadDormeur = new Thread(new
ThreadStart(dormir));
threadDormeur.Start();
Console.WriteLine(nom + " est
maintenant endormi !");
}
public void dormir(){
while (true){
Console.Write("RRRR.....PSSSS......");
Thread.Sleep(1000);}
}
public void seReveiller(){
threadDormeur.Abort();
Console.WriteLine("Bonjour, je
m'appelle " + nom + " et je suis réveillé !");
}
public void continueADormir(){
Console.WriteLine("Aujourd'hui, c'est
dimanche. Je reste couché !");
}
}
```

19

Exercices 19.1

1) Sauvegarde sur fichier plat à travers les *streams*
2) Sérialisation
3) Base de données relationnelles
4) Base de données orientée objet

Exercice 19.2

Selon la nature de ce que l'on sauve (nombre, string…) et la manière dont on le sauve (avec tampon ou sans tampon), plusieurs sous-classes de Stream peuvent être envisagées.

Exercice 19.3

Le filtre appose des additions aux « streams » selon la nature de ce que l'on sauve de manière à adapter la manière de le sauver. C'est l'application du design pattern Decorator.

Exercice 19.4

Car on ne sauve pas que l'objet en question mais également toute la série des objets référés par lui, et cela de manière enchaînée, d'objets en objets.

Exercice 19.5

Il ne sert à rien de sauvegarder certains objets qui n'ont de raison d'être que temporaire et dans un contexte d'exécution donné. C'est le cas par exemple des objets de type « Thread » ou « Stream ».

Exercice 19.6

La table se doit de posséder un attribut « clef » de manière à singulariser les enregistrements, alors qu'un objet est rendu unique par la place qu'il occupe dans la mémoire. Il en va de même pour la notion de clef étrangère (indispensable pour la réalisation des relations `1-n` en relationnel) et qui disparaissent de l'orienté objet.

Exercice 19.7

L'héritage entre deux classes peut très simplement se reproduire par une relation de type `1-1` entre les deux tables correspondantes.

Exercice 19.8

Par la nécessaire présence de la clef étrangère dans la table du côté n, on conçoit aisément qu'il est impossible de réaliser une relation de type `n-n`. Il faut la casser par une table intermédiaire qui contiendra les deux clefs étrangères des tables que l'on associe. Tout cela disparaît de l'orienté objet pour lequel, il suffit que les objets possèdent parmi leurs attributs un tableau de pointeurs vers tous les objets auxquels ils se trouvent associés. Et cela peut bien évidemment se faire dans les deux directions.

Exercice 19.9

Justement pour éviter de devoir se préoccuper des clefs et pour maintenir la relation d'héritage pour ce qu'elle est. Les bases de données OO réalisent la sauvegarde des objets de la manière naturelle dont ils sont utilisés dans les programmes OO.

Exercice 19.10

Ces deux extensions sont différentes, car elles résultent de deux dynamiques antagonistes, l'une trouvant son origine dans SQL, et tentant d'intégrer les modes relationnels propres à l'objet, et l'autre trouvant son origine dans l'OO et tentant d'étendre le langage SQL aux mécanismes d'usage et de requête classique des objets.

21

Exercice 21.1

Directionnels : le Web, les relations d'amitiés.
Symétriques : Internet, les réseaux routiers et biologiques (pour l'essentiel).

Exercice 21.2

La taille de la liste peut être quelconque et on peut introduire et supprimer des éléments sans que cela ne pose problème. La création et suppression de listes ne générera pas de zones inoccupées dans le support mémoire, car on peut entreposer les objets de la liste où l'on veut dans cette mémoire.

Exercice 21.3

```
import java.util.*;
class VectorInteger {
private Vector unVecteur;
public VectorInteger() {
unVecteur = new Vector();
}
public void
ajouteIntegerEnQueue(Integer i)
{
unVecteur.addElement(i);
}
public int afficherInteger (int
i) {
return
((Integer)unVecteur.elementAt(i
)).intValue();
}
public void
montreToutLeVecteur() {
for (int i=0;
i<unVecteur.size(); i++)
((Integer)unVecteur.elementAt(i
)).intValue();
}
}
```

Mais depuis les nouvelles versions de Java, il est bien évidemment beaucoup plus simple d'utiliser les generics : `ArrayList<Integer>`.

Exercice 21.4

Car chaque élément de la liste pointe vers le suivant et, en C++, tout pointeur nécessite l'utilisation explicite de `*`. En java, ce n'est pas nécessaire, car les objets sont toujours « pointés » par définition, aucun objet ne pouvant être utilisé « par valeur » ou sur la mémoire pile.

Exercice 21.5

La première liste liée reprend les nœuds du graphe et pour chacun, une deuxième liste liée permet d'indiquer les nœuds auxquels les premiers sont associés.

Exercice 21.8

Le code affichera : `6724975`.

Exercice 21.9

Le code affichera infiniment les deux chiffres `1,2`.