TEST 1 :

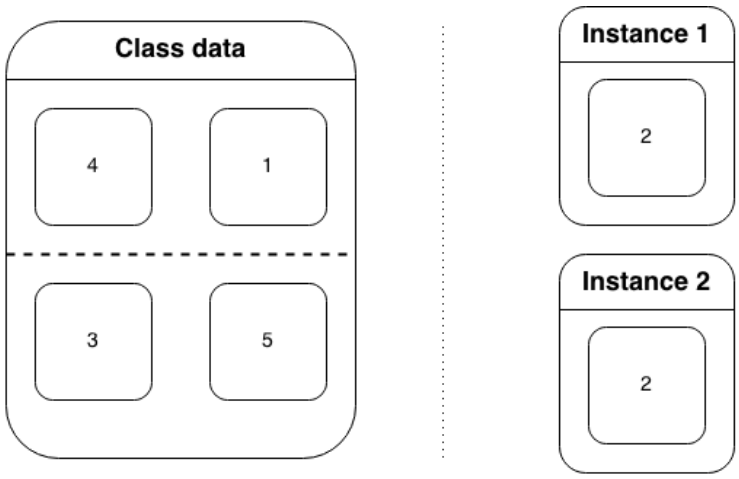
F	Il faut éviter d'encapsuler pour diminuer le couplage.
F	Les membres « private » sont visibles depuis d'autres classes externes sous certaines conditions.
V	De l'extérieur d'un package, impossible d'hériter d'une classe qui n'est pas « public ».
F	Une variable peut être uniquement de type primitif ou de classe.
F	Un package montre toutes les variables et méthodes « public » et « protected ».
F	Un héritier voit uniquement les membres « public » de son parent qui se trouve dans un autre package.
V	Une class est toujours dans un package.
V	Les instances n'apparaissent qu'au moment de l'exécution (run-time).
F	Un hériter dans le même package voit uniquement les membres « public » et « protected ».
V	Dans un diagramme de classe, le nom des classes abstraites est noté en italique.
F	On peut créer une instance de classe abstraite.
F	Dans un diagramme de classe : public = '+', private = '-', protected = '~ ', package friendly = '#'.
F	« string » est un type primitif.
F	Une class est accessible uniquement si elle se trouve dans le même package et qu'elle est « public ».
F	Seuls les attributs et les classes ont une visibilité.
V	Un package ajoute une couche d'encapsulation vis-à-vis de l'extérieur.
F	Il ne peut pas y avoir deux classes de même nom, dans des packages différents.
F	Un package ajoute une couche d'héritage vis-à-vis de l'extérieur.
V	Un package ne montre que les classes et interfaces « public ».
V	Il ne peut pas y avoir deux classes de même nom dans le même package.

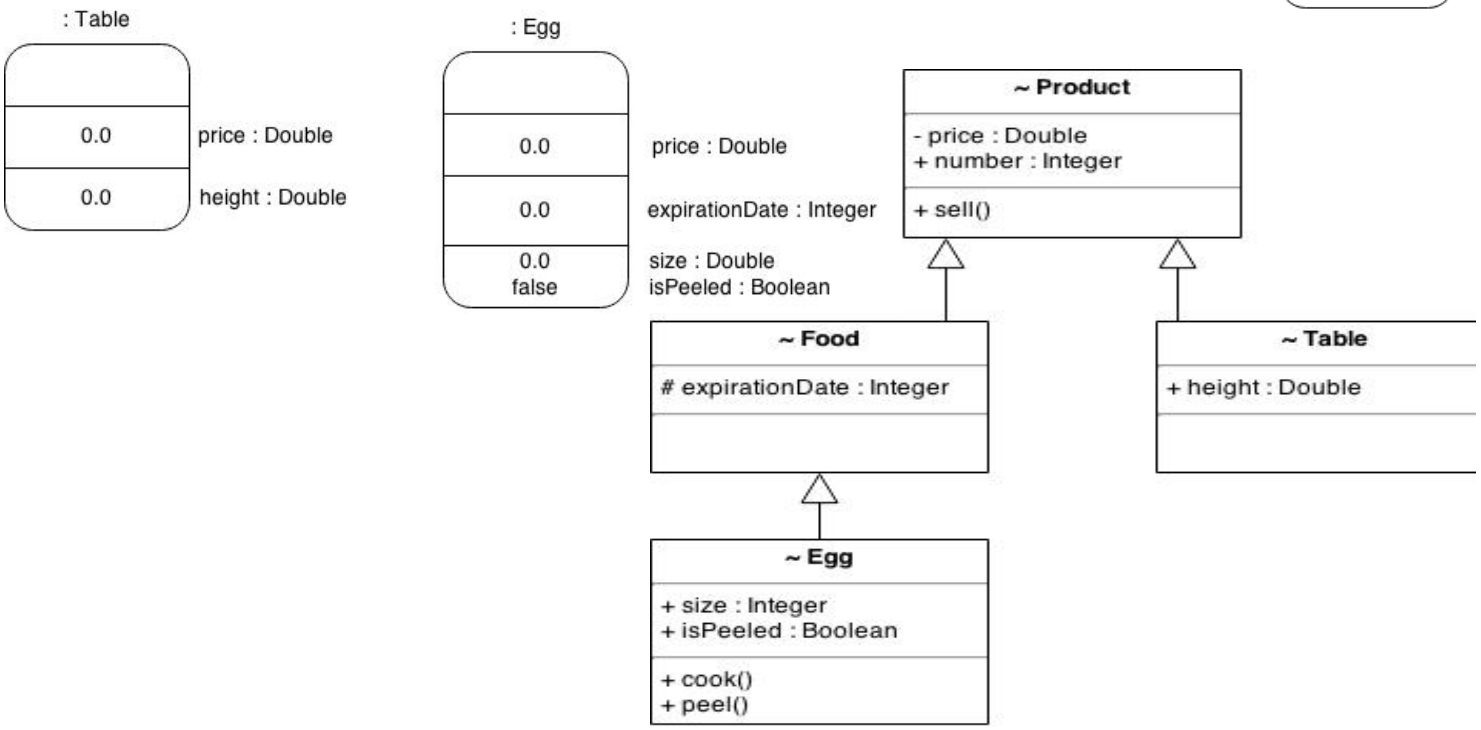
3. Si on est à l'extérieur d'un package et que la classe est non-publique, impossible pour nous de la voir et don de l'hériter.
4. Type primitif, classe et tableau.
5. Un package ne montre que les variables et méthodes « public » (ou « protected » mais seulement vis à vis de ses enfants).
6. Un héritier voit les membres publics **et** protected de son parent qui se trouve dans un **autre** package.
7. Même si on ne crée pas de package, Java en fait un par défaut.
9. Dans un **même** package, tout le monde voit les membres « public », « protected » et « package friendly », héritier ou pas.
12. protected = '# ' package friendly = '~'.
13. String est une classe mais a certains comportement comme les types primitifs ('' '' + …).
14. Dans le **même** package, on peut accéder aux classes qui sont « public » et « package friendly ».
15. Attributs, classes et méthodes.
16. Encapsuler == cacher l'information par rapport à l'extérieur. Si on met un package, on cache les classes « package friendly ».
18. Un package ajoute une couche d'**encapsulation**.
20. Java ne serait pas différencié laquelle on veut appeler.

F	Un constructeur n'a pas de type de retour, il retourne implicitement « void ».
F	Chaque classe dans la hiérarchie d'héritage d'une instance possède son propre « sac de données ».
V	Les variables d'instance sont toutes initialisées lors de la créations du « sac de données ».
F	Le « state » d'une instance est l'ensemble des valeurs des attributs de la classe.
V	Un constructeur peut avoir une visibilité « private »
V	La classe « Object » n'hérite pas de la classe « Class ».
V	Toute classe hérite de la classe « Object »
V	Dans un diagramme de classe, les membres « static » sont soulignés.
F	Une classe est chargée en mémoire à chaque appel d'une méthode de classe.
V	Il ne peut y avoir qu'une seule structure « class data » dans la mémoire pour une classe.

TEST 2 :

1. Class methods
2. Instance variables
3. Template for instance variables
4. Class variables
5. Instance methods





1. Le constructeur retourne « this ».
2. La classe qu'on instancie a son sac de données où les données des parents seront contenus.
3. Ensemble des valeurs des attributs de l'**instance**.
4. Voir le singleton.
5 – 6. « Tout est Object » => Toutes les classes héritent, par défaut, d'Object.
7. La classe est chargée une seule fois en mémoire, dès son premier appel.

Pour les sacs de données :

- Toutes les variables sauf static.
- Indiquer les valeurs par défaut.
- Aucun type de retour, ni de visibilité.
- Pas de méthodes.
- Ne pas oublier de montrer les différentes couches.
- Indiquer les nom des variables + type HORS du sac.
- « Lorsque les sacs ont été crées grâce à l'opérateur "new", on a donné le nom d'une classe. C'est la classe réelle du sac. C'est ça qu'on note au dessus du sac. »

TEST 3 :

V	Le type d'une instance définit l'ensemble des membres visibles.
F	Un constructeur n'a pas de type de retour, il retourne implicitement « void ».
F	Un constructeur ne peut pas avoir de visibilité « private » .
V	Une instance peut avoir le type de sa classe.
F	Le pattern singleton permet de distribuer plusieurs instances aux « utilisateur » .
V	Une classe ne peut être chargée en mémoire qu'une et une seule fois.
V	Une instance peut avoir le type d'une interface implémentée par son parent.
F	Une instance ne peut pas avoir le type du parent d'une interface qu'il implémente.
V	Le « static binding » concerne les méthodes de classe et s'effectue à la compilation.

```
class  P {
        public void p() { System.out.println("pp") ; }
        public static void print() { System.out.println("P") ; }
}

class  Q extends P {
        public void p() { System.out.println("qp") ; }
        public void q() { System.out.println("qq") ; }
        public static void print() { System.out.println("Q") ; }
}

class  R extends Q {
        public void p() { System.out.println("rp") ; }
        public void q() { System.out.println("rq") ; }
        public void r() { System.out.println("rr") ; }
        public static void print() { System.out.println("R") ; }
}

public class Main {
        public static void main(String[] args) {
                P p = new R() ;
                p.p() ;
                p.print() ;
                Q q = new Q() ;
                q.p() ;
                q.q() ;
                p.print() ;
                R r = (R) p ;
                r.p() ;
                r.r() ;
                r.print() ;
                r  = (R) q ;
                r.r() ;
        }
}
```

```
rp
P
qp
qq
P
rp
rr
R
ClassCastExcption
```

TEST 4 :

```
class  A {
        private static String s = ''Argh !'' ;
        public A() {
                System.out.println(''Waw !'') ;
        }

        {
                System.out.println(''Arf !'') ;
        }

        static{
                System.out.println(s) ;
        }
}
class  B extends A {
        {
                System.out.println(''Hey !'') ;
        }
        public B() {
                System.out.println(''Yo !'') ;
        }
}
class  C extends B {
        public C() {
                System.out.println(''Paf !'') ;
        }

        static{
                System.out.println(''Oups !'') ;
        }
        public static void main(String[] args) {
                C c = new C() ;
                D.print() ;
        }
}
class  D{
        static{
                System.out.println(''Ah !'') ;
        }

        {
                System.out.println(''Toc !'') ;
        }
        public static void print() {
                System.out.println(''Yaaa !'') ;
        }
}
```

```
Argh !
Oups !
Arf !
Waw !
Hey !
Yo !
Paf !
Ah !
Yaaa !
```

Au moment du chargement de la classe C, on va parcourir toute l'hiérarchie jusqu'à A. Une fois là, on exécute tout les <clinit>, du haut de l'hiérarchie jusqu'en bas. Une fois tout les <clinit> effectué, new C() . On remonte de nouveau dans la hiérarchie et on exécute tout les <init> + constructeur jusqu'à C.

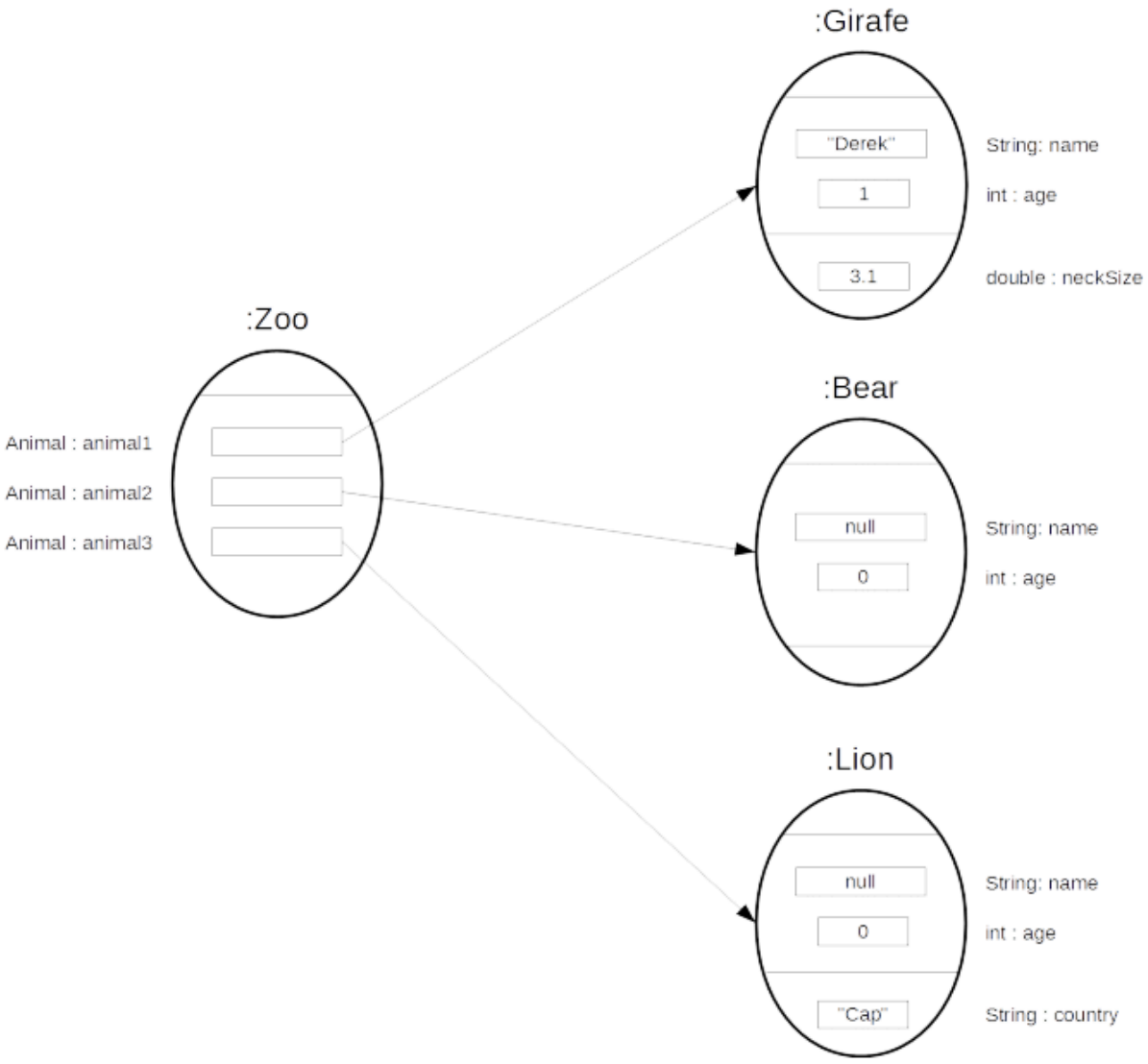
(Attention, si à la place de C c = new C() ; on avait B b = new B() ; le seul changement serait qu'on exécute pas le constructeur de la Classe C.)

Ensuite on charge la Classe D => Même que pour le chargement de la Classe C et on exécute la méthode de classe (static) print(), donc on ne passe pas par le constructeur!

TEST 5 :

```
class  Zoo {
        private Animal animal1, animal2, animal3;

        public Zoo(Animal animal1, Animal animal2, Animal animal3) {
                this.animal1 = animal1 ;
                this.animal2 = animal2 ;
                this.animal3 = animal3 ;
        }

        public static void main(String[] args) {
                Animal bear = new Bear() ;
                Animal girafe = new Girafe(''Derek'', 1, 3.1) ;
                Animal lion = new Lion(''Cap'') ;
                Zoo zoo = new Zoo(bear, girafe, lion) ;
        }
}

class  Animal {
        private String name;
        private int age;

        public Animal(String name, int age) {
                this.name = name;
                this.age= age;
        }
}

class Bear extends Animal {
}

class Lion extends Animal {
        private String country ;

        public Lion(String country){ //*
                this.country = country ;
        }
}

class Girafe extends Animal {
        private double neckSize;

        public Girafe(String name, int age, double neckSize){
                super(name, age) ;
                this.neckSize = neckSize ;
        }
}
```

* Par défaut, il existe un super() implicite, hélas le problème est qu'on a reecrit le constructeur vide d'Animal. De ce fait, on est obligé de rajouter super(String, int) si on ne veut pas d'erreur à la compilation (ou sinon de rajouter en Animal le constructeur vide).



TEST 6 :

F	La « frame » d'une méthode est conservée pour ne pas en créer une nouvelle à chaque appel.
F	La « method area » permet de stocker les instances (données) .
V	Deux méthodes d'instance « synchronized » peuvent être exécutées en même temps par deux Threads différents, sur des objects différents, mais de même classe.
F	Un constructeur n'a pas de type de retour, il retourne implicitement « void ».
V	La JVM utilise le « dynamic binding » pour détermine quelle méthode d'instance invoquer à l'exécution.
V	Chaque Thread possède son propre « stack ».
F	Pour qu'un objet soit « thread-safe », il doit être immuable.
F	Pour qu'un objet soit immuable, les attributs, les méthodes, et la classe doivent entre-autres être « final ».
V	Les blocs d'initialisation statiques sont invoqués automatiquement au chargement de la classe.
F	Il n'y à qu'un seul « PC register » dans la JVM.
V	Chaque exécution de méthode à sa « frame » dans le « stack ».
V	Un constructeur peut avoir la visibilité « private ».
V	Chaque classe chargée en mémoire est liée à un ODC dans le « heap ».
F	Deux méthodes de la classe « synchronized » d'une même classe peuvent être executées en mêm temps pas deux Threads différents.
F	Un objet dont le nombre de référence n'est pas nul ne peut surtout pas être supprimé par le « garbage collector ».

Class a = Class.forName(« Girafe ») ;
Girafe (ou Object) b = a.newInstance() ;
Method[] c = **a. getMethods()** ;
c[0].invoke(b) ;

TEST 7 :

F	Il faut réduire les dépendances abstraites entre les packages.
V	Une classe abstraite peut contenir des méthodes « concrètes ».
F	Deux méthodes de classe « synchronized » d'une même classe peuvent être exécutées en même temps par deux Threads différents.
V	Une classe abstraite peut contenir un constructeur.
F	Une interface peut contenir des méthodes « concrètes ».
V	L'appel à un constructeur est une dépendance concrète.
V	Deux méthodes d'instance « synchronized » peuvent être exécutées en même temps par deux Threads différents, sur des objets différents, mais de la même classe.
F	Une interface est la déclaration d'un ensemble de méthodes statiques et/ou de constantes.
V	Une interface peut contenir des méthodes abstraites.
F	Une interface peut contenir un constructeur.

```
package Tv ;

interface Tv { //3
        void on(Client user) ; //4
}

public interface TvUser {
        void callback() ;
}

public class SamsungN520 implements TV { //1
        public void on(Client user) { //5
                user.callback() ;
        }
}

package client ;

class Client impements TvUser {
        public static void main(String[] args) {
                SamsungN520 maTv = new SamsungN520() ; //2
                Client moi = new Client() ;
                maTv.on(moi) ;
        }

        public void callback() {
        }

}
```

1. Il faut réduire les dépendances **concrètes**.
4. On ne pourra pas faire de new sur le constructeur, mais les classes enfants l'utiliseront lors de leurs instanciations.
8. Une interface ne peut pas avoir de méthodes statiques/de classe.
9. Une interface ne possède que des méthodes abstraites.

Erreurs : 1. Le but est de cacher cette classe par une interface. En mettant le public, nous perdons cette logique.

2. On veut être complètement indépendant de l'implémentation de la classe SamsungN520 et donc nous devrions plutôt utiliser l'interface qu'elle implémente.

3. Si notre interface n'est pas publique, elle ne nous sert à rien, car à l'extérieur du package, personne ne pourra la voir.

4 – 5. Comme la 2.

TEST 8 :

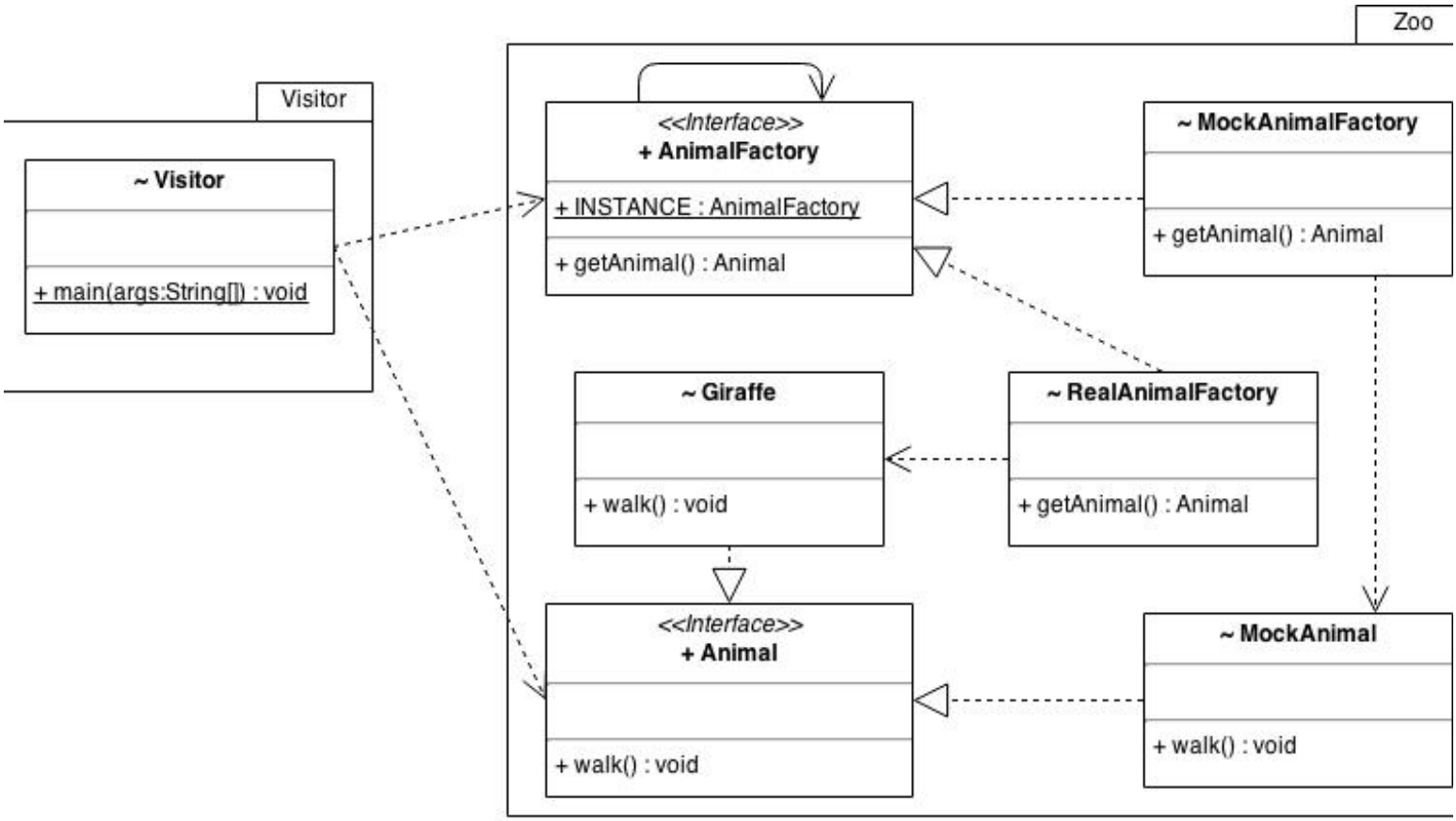
```
package tv ;

public interface Tv {
	void on() ;
}

public class SamsungN52 implements Tv { //1

	@Override
	public void on(){
	}

	public void off(){
	}

}

public class SonyBravia53 implements Tv { //2 + 5
}

public interface TvFactory {

	static TvFactory INSTANCE = new TvFactoryImpl() ;

	Tv getTv() ;
}

public class TvFactoryImpl { //3 + 6

	@Override
	public Tv getTv() {
		return new SamsungN52() ;
	}
}

//-----------------------------------------------------------------------------------------------------

package client ;

import tv.Tv ; //7

public class Client { //4
	public static void main(String[] args){
		Tv tv = TvFactory.getTv() ; //8
		tv.on() ;
	}
}
```

1-2-3-4. Le but est cacher ces 4 class par des interfaces pour protéger les Objets, en les mettant public, cette logique est perdue.
5. Vu que la classe SonyBravia53 implémentes la classe Tv, elle doit obligatoirement implémenter la méthode on() ;
6. Cette classe doit implémenter l'interface TvFactory si on veut être capable de la cast en TvFactory.
7. Il manque un import de la classe TvFactory.
8. Si on veut avoir accès à cette méthode, on a besoin de la variable static INSTANCE -> TVFactory.INSTANCE.getTv() ;

TEST 9 :

```
package zoo ;

public interface Animal{

	void walk() ;
}

class Giraffe implements Animal {

	@Override
	public void walk() { }
}

class MockAnimal implements Animal {

	@Override
	public void walk() { }
}

public interface AnimalFactory {

	AnimalFactory INSTANCE = new MockAnimalFactory() ;

	Animal getAnimal() ;
}

class RealAnimalFactory implements AnimalFactory{

	@Override
	public Animal getAnimal() {
		return new Giraffe() ;
	}
}

class MockAnimalFactory implements AnimalFactory{

	@Override
	public Animal getAnimal() {
		return new MockAnimal() ;
	}
}

//-----------------------------------------------------------------------------------------------------

package visitor ;

import zoo.Animal ;
import zoo.AnimalFactory ;
```

```
class Visitor {
        public static void main(String[] argd){
                Animal animal = AnimalFactory.INSTANCE.getAnimal() ;
                animal.on() ;
        }
}
```



TEST 10 :

V	Une Factory peut avoir plusieurs implémentations possibles, on choisira la bonne lors la compilation ou lors de l'exécution.
F	Par défaut, une classe est « public ».
V	Une constante d'interface peut contenir la référence à une instance.
F	Par défaut, une méthode d'une interface est toujours « public static abstract ».
F	Si la classe « Parent » hérite de la classe « Ancêtre », l'opération de cast automatique lorsqu'on cast un objet de type « Ancêtre » vers un type « Parent ».
F	Les constantes d'une interface sont initialisées à la compilation.
V	Par défaut, un attribut d'une interface est toujours « public static final ».
F	L'injection de dépendances permet à un client de récupérer un « Service Object » auprès d'une Factory.
F	Pour importer toutes les classes et interfaces d'un package il faut ajouter la ligne « import monpackage. ? »
V	Un objet immuable est toujours Thread-safe.
V	« TvFactory INSTANCE = new MockTvFactory() ; » a une dépendance concrète.
V	Un fichier « properties » peut être chargé dans une « Map<String, String> » grâce à l'instruction « load() » ;
F	Le polymorphisme est l'idée de protéger l'information contenue dans un objet.
F	Le polymorphisme se base sur le « static binding ».
F	Un « Service Object » est une classe dont tous les membres sont « static ».
V	Si la classe « Parent » hérite de la classe « Ancêtre », l'opération de cast est automatique lorsqu'on cast un objet de type « Parent » vers un type « Ancêtre ».
V	Un objet stateless est toujours Thread-safe.
V	L'utilisation d'une Factory permet d'encapsuler la logique d'instanciation et de ne plus utiliser le mot clef « new » pour certaines classes.
F	Une interface peut contenir un constructeur.
V	L'introspection permet d'instancier n'importe quelle classe, même les classes « package friendly » des autres packages .

1. Compilation => Injection de dépendance, Execution => Plugin.
2. Par défaut, une classe est package friendly.
4. Une méthode est « public abstract ».
5. Suffit de changer Ancêtre par Object.
6. Fait au chargement de la classe non pas à la compilation.
8. Le client le récupère pas, mais plutôt quelqu'un d'autre le fait pour lui (le main, par exemple, qui après l'insérera bien avec le constructeur, bien avec le setter).
9. « import monpackage.* ».
11. Le polymorphisme est l'idée d'utiliser un même code