

Révision Java

Quiz 1 - Getter & Setter :

“**accessor**” methods -> getter

```
public int getAge(){  
    return age;  
}
```

“**mutator**” methods->setter

```
public void setName(String nom){  
    this.nom = nom;  
}
```

Par quoi peut t'on remplacer XXX ?

```
package toto;  
XXX class Foo{}
```

Réponse : **public** & Rien, **package private**

Quiz 3 - final & methods:

En gros : changer une valeur constante (final en java) = erreur

Attention cela ne fonctionne pas pareil pour les Liste & les tableaux

```
class Toto{  
    void meth(){  
        foo.doSomething();  
    }  
    //other code  
}
```

Ici on ne sait pas si **doSomething** est déclaré dans Toto

Contrairement a :

```
class Toto{  
    void meth(){  
        doSomething();  
    }  
    //other code  
}
```

Ici doSomething est déclaré dans Toto

Quiz 4 - Lambda

```
public class Main {  
  
    public static void main(String[] args) {  
        String[] keywordList={"Oculus", "Eau", "Eau", "Feu"  
                                , "Yahourt", "Chaise",  
                                "Takenoko", "Semestre 5",  
                                "Chien", "Chat", "Balais"}; //Ajouts des mot clé dans  
un tableau  
        //Exemple d'utilisation des lambda :  
        // Exemple 1 : Récupérer dès le 1er mot avec un Y  
        // Ici on filtre les mot qui contiennent que y et on  
retourne le 1er résultat  
  
        System.out.println(Arrays.stream(keywordList).filter(keyword ->  
keyword.contains("Y")).findFirst().get());  
        //Exemple 2 : Nombre de mot avec des a  
        System.out.println(Arrays.stream(keywordList).filter(keyword  
-> keyword.contains("a") ).count());  
        //Cela marche évidemment avec une liste aussi  
        List<String>keywords=new ArrayList<>();  
        keywords.addAll(Arrays.asList(keywordList));  
        //l'on peut aussi parcourir avec un forEach  
        keywords.forEach(o -> System.out.println(o));  
        //On peut supprimer des élément de cette liste a une  
condition  
        //ici on supprimer tous les éléments qui contiennent des a  
        keywords.removeIf(o -> o.contains("a"));  
    }  
}
```

Ceci est un résumé très rapide

A voir aussi :

Utilisation de l'option map

<https://www.mkyong.com/java8/java-8-streams-map-examples/>

Quiz 8 - Exception

Quand l'on fait un `throw MonExceptionException` dans une méthodes, il faut aussi l'indiquer en "haut" en mettant **throws**. L'ajout de **throws** permet d'indiquer que les exception susceptible de se déclencher

```
public class YourTeacher extends ActionHero{
    public void swim() throws MonExceptionException {
        if(true){
            throw new MonExceptionException();
        }
        System.out.println("swim");
    }
}
```

RUNTIME EXCEPTION :

Ce qui me surprend continuellement, c'est le nombre de développeurs Java que j'interroge et qui n'ont aucune idée de ce qu'est une `RuntimeException` ou comment elle peut être utilisée. Les `RuntimeException` ont le même objectif que les `checked Exception`. communiquer des conditions exceptionnelles (échecs inattendus, etc.) à l'utilisateur. Ils peuvent être lancés et capturés comme des exceptions vérifiées. Toutefois, la gestion des exceptions d'exécution **n'est pas appliquée par le compilateur**.

src : <http://johnpwood.net/2008/04/21/java-checked-exceptions-vs-runtime-exceptions/>

"If you want to write a runtime exception, you need to extend the `RuntimeException` class."

src : https://www.tutorialspoint.com/java/java_exceptions.htm

```
public class MonExceptionException extends RuntimeException {
}
```

- **Unchecked exceptions – Subclass of `RuntimeException`**
- **Checked exceptions – Subclass of `Exception`**

`RuntimeException` is a subclass of `Exception`

"If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception"

src : <https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/>

INHERITANCE:

STATIC TYPE :

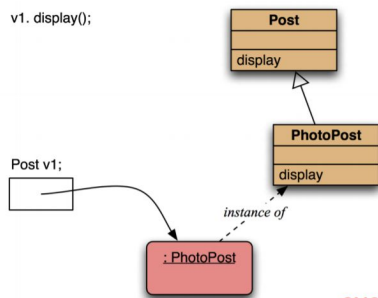
`Car c1 = new Car();` Will be of type `Car` at compile time (static) and type `Car` at runtime (dynamic).

`Vehicle v1 = new Car();` Will be of type `Vehicle` at compile time (static) and type `Car` at runtime (dynamic).

OVERRIDING:

Définir les fonctions à la fois dans les super et subClasses

METHOD LOOKUP:



Si on n'avait pas de méthode display dans photoPost, la méthode display de Post sera utilisé : Héritage mais pas d'overriding.

INSTANCEOF :

permet d'identifier le type dynamique.

permet de vérifier si un objet est une instance d'une classe spécifique.

```
public void doSomething(Number param) {  
    if( param instanceof Double) {  
        System.out.println("param is a Double");  
    } else if( param instanceof Integer) {  
        System.out.println("param is an Integer");  
    }  
}
```

TOSTRING:

```
System.out.println(monObjet); // invokes monObjet.toString()  
// si monObjet.toString n'existe pas, la fonction toString de la classe object  
est appelée
```

EQUALS AU LIEU DE == !!!

From courses : (Nathan was here)

Code source (ce qu'on écrit) --- (Compilateur) --> Byte Code ---- (JVM) ----> Objet en interaction

A String is immutable

String dans java.lang; List dans java.util

foreach != for : variables locales dans foreach

Toutes collections ne peuvent pas stocker de primitifs

== : test d'identité

.equals() : test d'égalité

Iterator équivalent aux pointeurs en C

Lambda :

Toutes les écritures suivantes sont équivalentes :

```
list.forEach((Element element) → {  
System.out.println(element.getName());});  
list.forEach((element) → { System.out.println(element.getName());});  
list.forEach(element → { System.out.println(element.getName());});  
list.forEach(element → System.out.println(element.getName()));
```

Interfaces :

Toutes les méthodes sont public.

Tous les attributs sont public, static, final.

Une enum peut **implements** mais **NE PEUT PAS extends**

Une classe peut être public ou package-private

Correction du DS de 2015

LIEN DU DEVOIR : <https://polytechsi3.files.wordpress.com/2016/01/pooreview.pdf>

Attention correction fait maison - IL PEUT Y AVOIR DES ERREURS

La classe ci-dessous :

QUESTION 1 :

- a) FAUX, pour sous classé (**extends**) une classe il faut que son constructeur soit package private, protected ou public.

- b) **VRAI OU FAUX**, elle ne peut pas s'instancier car son constructeur est privé mais dans la classe CiDessous elle-même, on peut utiliser son constructeur

ex :

Ici pas de soucis dans la classe elle-même

Par contre ici, on ne peut pas

QUESTION 2 :

- a) **FAUX**, on ne peut pas créer de **sous classe** a partir d'une **final class**
Petit plus, les méthode qui sont final ne peuvent pas être @Override
- b) **VRAI**, elle peut être instancier même si elle n'a pas de constructeur, quand il n'y a pas de constructeur, le constructeur par défaut est utilisé ici cela équivaut a ce constructeur :

```
public CiDessous(){  
  
}
```

QUESTION 3 :

- a) **VRAI**, une classe abstraite peut être sous classé.
- b) **FAUX**, une classe abstraite ne peut être instanciée.

QUESTION 4 :

- a) La ligne 8 sert à indiquer que l'on @Override une méthode déjà existante
En gros, on redéfinit une méthode de la classe mère, encore une fois **si une méthode de la classe mère est final, ON NE PEUT PAS L'@OVERRIDE.**
- b) Pour les lignes 10 à 12, on vérifie si l'objet n'est pas exactement le même, en gros

```
CiDessous c = new CiDessous();  
if(c==c){  
    System.out.println("vrai");  
}else{  
    System.out.println("faux");  
}
```

Ici, le code va retourner vrai CAR IL S'AGIT **EXACTEMENT** du même objet
alors que ci-dessous il va retourner **FAUX**

```
CiDessous c = new CiDessous();  
CiDessous d = new CiDessous();  
if(c==d){  
    System.out.println("vrai");  
}else{  
    System.out.println("faux");  
}
```

Même si les objet sont égaux au niveau des valeurs le "==" entre deux objet va vérifier si il s'agit de la même instance

- c) Les lignes 13 à 15 vérifie si l'objet mis en paramètre est un objet de type avec **instanceof** Person, s'il ne l'est pas, il retourne false.
- d) Les lignes 16-17, on cast l'objet en paramètre du même type que l'objet qui fait appel au .equals() et on vérifie qu'ils ont la même valeur entre leurs attributs et l'on retourne **TRUE** si leurs attributs sont égaux.

Question 5 :

a)

A :

```
if(toqsik()>0)
    //DoSomething
else
    //DoSomethingElse
```

B :

```
try{
    int i = toqsik();
}catch(QuaboomException e){
    //DoSomething
}
```

b)

A:

Avantages :

Ne force pas le développeur à gérer l'erreur (pas sûr de moi)

Désavantages :

N'arrête pas le programme s'il y a une erreur.

Oblige à avoir une valeur de retour correspondant à l'erreur

Le développeur doit savoir que -1 correspond à l'erreur, alors qu'une exception est beaucoup plus explicite.

B:

Avantages :

Les erreurs ne peuvent pas être ignorées.

Pas de valeur de retour spécial.

Peuvent être gérées avec des catch.

Gestion de multiples exceptions

Désavantages :

Force à gérer l'exception même si celle-ci est mineur.

Question 6 :

- a) **FAUX**, car la méthode CanSwim n'est pas public dans YourTeacher alors qu'elle vient d'une interface, ici elle est en package private.

Question 7 :

- a) Instanceof n'est pas orienté objet, Notez que si vous devez utiliser cet opérateur très souvent, c'est généralement un indice que votre design a quelques défauts. Donc, dans une application bien conçue, vous devriez utiliser cet opérateur aussi peu que possible (bien sûr, il y a des exceptions à cette règle générale). (merci stackoverflow)
- b) **(Cours du prof)**


```

public class PetOwner {
    public void talking(Object obj) {
        if (obj instanceof Hamster) {
            Hamster hamster = (Hamster) obj;
            hamster.talk();
        } else if (obj instanceof Axolotl) {
            Axolotl axolotl = (Axolotl) obj;
            axolotl.talk();
        }
    }
    public static void main(String[] args) {
        PetOwner pwner = new PetOwner();
        pwner.talking(new Hamster());
        pwner.talking(new Axolotl());
    }
}

```

- N'utilisez pas le type d'exécution d'un objet pour déterminer quel code doit être exécuté lorsque vous appelez l'une de ses méthodes;
- utilisez le **polymorphisme** et laissez l'objet décider du code à exécuter.
Parfois, cependant, vous devez déterminer le type d'exécution d'un objet afin de déterminer ce que votre classe va faire.

Réponse b :

Le code ci-dessus pourrait être amélioré en faisant une **interface** Animal avec la méthode **void talk()**; cela donnerait :

```

public class PetOwner {
    public void talking(Animal obj) {
        obj.talk();
    }
    public static void main(String[] args) {
        PetOwner pwner = new PetOwner();
        pwner.talking(new Hamster());
        pwner.talking(new Axolotl());
    }
}

```

Question 8 :

a.

```
instanceMethod() in Bar  
classMethod() in Foo
```

b. Il est possible d'appeler une méthode static à partir d'un objet. Donc pas d'erreur à l'exécution.

Question 9 :

a. Il faudrait vérifier que les paramètres dans le constructeur sont bien donnés, sinon on renvoie une erreur.

b.

```
public Person(String name, int age) {  
    if(name == null) {  
        throw new IllegalArgumentException("Name cannot be null");  
    } else if(name.equals("")) {  
        throw new IllegalArgumentException("Name cannot be empty");  
    } else if(age < 0) {  
        throw new IllegalArgumentException("Age cannot be negative");  
    }  
    this.name = name;  
    this.age = age;  
}
```