**-Héritage** : les champs private de la mère ne sont pas accessible directement par la fille, il faut un getter ou super.champs

-Le main est toujours static car cela permet son invocation automatiquement a l’exécution sous création d’un objet de la classe test.

-une meth static peut accéder uniquement a des membres static de sa class et n’a pas accès a this.

-this doit être le premier dans le constructeur s’il est présent.

-Meth d’instance peut accéder a tous les membres de sa classe (static ou non)

-**surcharge :** même type de retour, même nom et des paramètres différents.

-**Redéfinition** : même signature et corps différent. Le type de retour peut changer mais il doit être sous type du 1er.

-On peut changer les modificateurs d’acces lors de la **redéfinition** mais pour élargir :

Private<package< protected< public.

-meth final ne peut pas être redéfinie.

**-Initialisation se fait par 3 parties :**

-Par défaut : tout a zéro et a null.

-Bloc initialiseurs.

-Corps du constructeur.

- Type getP(return p ; ) -> violation d’encapsulation car p est une référence, du coup on pourra le modifier dans une autre classe.

-Super class = sous class

- type déclaré ne change pas, l’effectif si.

-get class retourne le type effectif.

-S’il y’a **redéfinition** des meth, c’est le type effectif qui détermine les meth à utiliser.

-Avec les méthodes static et les variables ca ne change pas, ceux du déclaré qui sont utilisés.

**-passage de paramètres** :

-les changements sur des pametres de type premitif n’affecte pas les variables du main.

-les changement sur les paramètres reference affecte les variables du main.

-les changements sur les references naffectent pas le main. Exemple

Deplace( Personne p){ p = null ;}

modifierAge(Personne p){p.age ++} ;

Main(){Personne p  =new Personne(20) ;….

Deplace(p) ;

print(p.age) ; 🡪OK p nest pas null

p.modifierAge(p) ;

print(p.age)🡪 21

-**Classe Membre** : un objet de la classe membre peut être créé uniquement par un objet de la classe englobante en utilisant juste new ClasseMembre()

-Lobjet de la classe englobante est accessible par l’objet de la classe interne :ClasseEnglobante.this

-une classe membre non static ne peut avoir des méthodes/champs static.

- la classe membre a accès au champs de la classe englobante même si sont private …

- l’englobante a accès au champs/méthodes de sa classe membre même si private.

-si une classe membre est visible hors sa classe englobante elle est dénotée ClasseEnglobante.ClasseMembre

Public class F{ public class H{..}}

Main; F a= new F(); F.H=a.new H();

A condition que H soit public.

-**classe membre static** a accès uniquement au champs static de la classe englobante.

-main : public class A{ public static class B{..}}

A.B nom = new A.B(); (condition B soit public )

- classe static a accès aux champs non static des classes membres private de la classe englobante.

-pour accéder a un champ de la classe englobante on utilise ClassEnglobante.this.champs

-**classe local** définie dans un bloc de code, visible uniquement dans ce bloc.

- elle a accès aux champs de la classe englobante et le champs du bloc englobant (ceux du bloc doivent être final ou effectively final, elle a accès au param passés a une méthode mais elle ne peut pas initialiser ces champs avec ces valeurs)

**Up ?action1 :action2** -> si up est true alors on fait action1 sinon action2.

-**classe anonyme** est une classe locale sans nom.

- créer un objet de classe anonyme qui hérite une classe de base : NomClassBase nom= new ClassBase(paremetre de base){

Champs de plus et méthodes de plus}

-elle n’a pas de constructeur.

-this dans l’expression lambda fait référence a l’objet de la classe englobante et non l’objet de la classe anonyme créé.

-expression lambda pour inverser un tableau :

TabOperation t= (int[] t)->{ inti nv[] =new int[t.length]; for(i=0;i<length,i++){inv[i] = tab[length-i-1]; } return inv}

Somme: fonctionnelle< Integer> addition = (a, b) -> a + b;

**Classes abstraites** contiennent au moins 1 meth abstraite-> on peut pas créer des objets de cette classe.

-Les interfaces n’ont que des champs constants et la plupart des méthodes sont abstraites et sont automatiquement public –> on déclare des objets de typer interface mais on les instancie pas.

-on peut avoir des méthodes static qui doivent avoir du corps, et aussi des méthodes private qui seront utilisées par les autres méthodes de l’interface. Les 2 sont pas accessibles par les classes qui implementent linterface

-il y’a des méthodes default, les classes qui implémentent l’interface peuvent ne pas les redéfinir.

- B extends A implements I1 : I1 contient f() et A aussi -> B contiendra celle de A(sa mère)

- B implements i1, I2 -> I1 contient f() (definie) et I2 contient f() (non definie) -> B doit redefinir f()

-une interface peut extends 2 interfaces.

-Methode generique peut appartenir a une classe genirique et non generique.

-On utilise extends pour la bornes des classes et interfaces a la fois. Tous les types extends Object.

-T peut être borné par plusieurs interfaces mais par 1 seule class(on mentionne la classe en 1er)

-Si A est un sous type de B alors c<A> NEST PAS un sous type de c<B>. exemple : Student extends Personne mais arrayList<Student> nest pas sous type de arrayList<Personne>.

-Par contre A[] t1 est sous type de B[] t2

- ? est utilisé quand aucun autre type ne dépend de lui, car il ne doit pas être déclaré

- ? peut être aussi borné mais par 1 seule borne :

? extends machin, ? super machin

-si B extends A : c<B> extends c< ?extends A> et

C<A> extends c< ?super B>

-**Il ya deux types d’exceptions** :

-checked : heritent de Exception

-Unchecked : heritent de RunTimeException

-On utilise throws devant les meth qui soulevent une exception checked.

-Si la meth la capture avec try … catch pas besoin de throws.

-Si une exception est captuée avec try catch (sans qu’on fasse throw a l’intérieur du bloc catch) le programme continuera l’exécution.

-sil y’a throw a l’interieur du bloc catch on execute le finally puis on souleve l’exception.

-sil ya return au bloc catch on execute d’abord le finally puis on retourne.

-si aucun catch ne correspond a l’exception on exécute le finally puis on affiche thread main …

-les blocs catch doivent être classés du plus spécifique au plus général.

-**swing** est développé en utilisant le design pattern MVC : model view controller.

-Il existe 4 concept en poo : heritage, encapsulation, polymorphisme et abstraction

**TD01** : public class **Personne** { private String nom; private String prenom; public int age; public Personne(String nom, String prenom, int age){ this.nom = nom; this.prenom = prenom; this.age = age; } public void setPrenom(String p){ this.prenom = p; } public void anniversaire(){ this.age ++; } public String toString(){ return "Je m’appelle : " + this.prenom + " " + this.nom + ". J’ai " + this.age + " ans."; 1 } } public class Test { public static void **main(String[] args){ Personne tony = new Personne("Parker","Tony",29); System.out.println(tony);//Tony jai 29 ans**

**Personne mickael = tony; mickael.setPrenom("Mickael"); System.out.println(tony);//Mickael jai 29 ans**

**//équivalent System.out.println(tony.toString());**

public class **A** { private int attr; public A(int value\_attr) { this.attr = value\_attr; } public boolean egal(A b) { return (this.attr == b.attr); } public int getAttr() { return this.attr; } public String toString(){ return "attribut:"+attr+" "; } **public static void main(String[] args) { A obj = new A(2); A obj2 = obj; A obj3 = new A(2);**

**if (obj.egal(obj2)) //-> egal**

**println("Egal"); else println("Different"); println((obj.egal(obj2)) ? "Egal" : "Different"); /egal/**

**println((obj2.egal(obj3)) ? "Egal" : "Different");/egal/**

**println((obj.egal(obj3)) ? "Egal" : "Different"); /egal/**

**println((obj == obj2) ? "Egal" : "Different"); /egal/**

**println((obj == obj3) ? "Egal" : "Different"); /different/**

**println((obj2 == obj3) ? "Egal" : "Different"); /different/**

**println(obj.toString()); /2/println(obj); //2**

**TD02:**

public class A { public static int a = 3 ; public int b ; public A ( int c ) { this.b = c ; 7 }

public void g ( ) { a = a+1 ; 11 b = b+1 ; }

public class Test { public static void main ( Strin g [ ] args ) { A u = new A( 0 ) ; A v = new A( 0 ) ; u . g ( ) ;

println( " u : a=" + u . a + " ; b=" + u . b ) **; 4 -1**

v . g ( ) ;

print( " v : a=" + v . a + " ; b=" + v . b ) ; **5-1**

u . g ();

print( " u : a=" + u . a + " ; b=" + u . b ) ; **6-2**

public static C h (C k ) {

k = new C( 5 ) ;return k;}

public static void main () {

C k = new C( 0 ) ;C l = *h* ( k )

println( k”+”l); **->0+5**

**Td 04**: Le constructeur de la super classe est appelé automatiquement même sans super.

**public class A** {

public void f (A x ) {

println ( "A. f (A) " ) ;}

public void g ( ) {

f (new A ( ) ) ;}

**public class B extends A** {

public void f (A x ) {

println ( "B. f (A) " ) ;}

public void f (B x ) {

println ( "B. f (b) " ) ;

A a = new A ( ) ;B b = new B ( ) ;

A c = new B ( ) ;

a . g ( ) ; ->A.f(A)

b . g ( ) ; ->B.f(A)

c . g ( ) ; ->B.f(A)

a . f ( a ) ;-> ->A.f(A)

a . f ( b ) ; ->A.f(A)

a . f ( c ) ; ->A.f(A)

b . f ( a ) ; ->B.f(A)

b . f ( b ) ; ->B.f(B)

b . f ( c ) ; ->B.f(A)

c . f ( a ) ; ->B.f(A)

c . f ( b ) ; ->B.f(A)

c . f ( c ) ; ->B.f(A)

**Td6** : On ne peut instancier ni une interface ni une classe abstraite.

-les classes abstraites ont des constructeurs, mais non pas les interfaces.

- A a = new B() OK -> si A est abstract, B non Abstract et A extends B

- A a = new B() : si A est une interface et B implements A🡪 OK

-Interface ont des attributs constantes(public static final) et doivent être initialisés. Les classes abstraites c’est comme les classes, pas de restriction

-on peut avoir des méthodes static abstract dans les classes abstraites. Dans les interfaces on peut avoir des abstract/ des static mais pas static abstract.

**Td08** : -Il faut catch d’abord les classes filles puis les classes mères sinon erreur a lexecution.

-Finally s’execute meme sil y’a erreur dans le bloc catch().

class A { int c = 0; public int getC () { return c ; } } class B extends A { int c = 1; } public class Test { public static void main ( String args []) { B b = new B () ; A a = b ; System . out . print (b . getC () +" "); System . out . println (a. getC () ); }

* 00

public class A {} public class B extends A {}

🡪toutes les methodes de la classe A sont heritees par la classe B, → la classe B peut d´eclarer une methode avec la mˆeme signature qu’une methode de la classe A

XXXX squaredLength = YYYY ; String s = " hello "; System . out . println ( squaredLength . applyAsInt ( s) );

🡪ToIntFunction<String> squaredLength = s -> s.length()\*s.length();

**public class Carte** { int numero ; public Carte (int numero ) { this . numero = numero ; } } **public class Individu** { boolean femme ; int anneeNaissance ; int moisNaissance ; public Individu ( boolean femme , int an , int mois ){ this . femme = femme ; this . anneeNaissance = an ; this . moisNaissance = mois ; } }

🡪Function < Individu , Carte > nouvelleCarte = ( bob -> { String aux =""; if( bob . femme ){ aux +="2"; } else { aux +="1"; } aux += Integer . toString ( bob . anneeN ); aux += Integer . toString ( bob . moisN ) ; return new Carte ( Integer . parseInt ( aux ) ); }) ;

🡪public static void echange ( T [] a , int i , int j) { T tmp = a [i ]; a[ i] = a[ j ]; a[ j] = tmp ; }

public class A {} public class B extends A {}

🡪→ ArrayList est un sous-type de List,

× ArrayList **est un sous-type de ArrayList,**

**× ArrayList est un sous-type de ArrayList,**

**→ ArrayList est un sous-type de ArrayList.**

public class Voiture { public int kmParcourus ; } public class Ferrari extends Voiture { }

🡪x Pour pouvoir creer des voitures et des Ferraris neuves (c’est-`a-dire qui n’ont parcouru aucun kilometre) il faut ajouter un constructeur dans les deux classes.

× On peut ajouter un constructeur dans Voiture qui construit des voitures avec un kilometrage non nul, sans ajouter de constructeur dans Ferrari.

→ On peut ajouter un constructeur dans Ferrari qui construit des Ferraris avec un kilometrage non nul, sans ajouter de constructeur dans Voiture.

× On peut ajouter un constructeur dans les deux classes, uniquement sous r´eserve que les deux aient les memes arguments.

Lorsqu’une classe A ´etend une classe B, est-ce que :

→ toutes les instances de A sont aussi des instances de B

× toutes les instances de B sont aussi des instances de A

× les deux r´eponses pr´ec´edentes sont vraies × les trois r´eponses pr´ec´edentes sont fausses.

class Portable { public void sonne ( Sonnerie s ) { s . sonne () ; } }

class Sonnerie { public void sonne () { System . out . println ("Z"); } }

class DoubleSonnerie extends Sonnerie { public void sonne () { System . out . println ("Z Z") ; } }

public class Bruit {

public static void main ( String [] args ) { Portable tel = new Portable () ;

Sonnerie s1 = new Sonnerie () ; DoubleSonnerie s2 = new DoubleSonnerie () ; Sonnerie s3 = new DoubleSonnerie () ; tel . sonne ( s1 ) ; tel . sonne (( Sonnerie ) s2 ) ; tel . sonne ( s2 ) ; tel . sonne ( s3 ) ; } }

→ Z [retour a la ligne] Z Z [retour a la ligne] Z Z [retour a la ligne] Z Z

public class A { public class B {} }

→ A.B b = a.new B(), avec a une instance de A