**Q1)** Quel spécificateur d'accès limite la visibilité d'un membre à sa seule classe ?

**a) public b) package c) protected d) private**

**Q2)** Choisissez l’expression qui permet de convertir le contenu d’un String nommé text en une valeur de type float

**a) text.floatValue() b) Float.parseFloat(text) c) String.parseString(text) d) (float)text**

**Q3)** Sélectionnez la seule phrase qui est vraie.

**a) Une classe abstraite ne peut pas être instanciée**

**b) Une classe abstraite ne peut pas définir des méthodes concrètes**

**c) Une classe abstraite ne peut pas définir des méthodes abstraites**

**d) Une classe abstraite ne peut pas définir des attributs**

**Q4)** Sélectionnez le bout de code que l’on utilisera pour une classe qui fait partie du package tools et qui utilise des classes des packages java.lang et java.util.

**a) import java.util.\*; import java.lang.\*; package tools;**

**b) import java.util.\*; import java.lang.\*;**

**c) package tools; import java.util.\*;**

**d) package tools; import java.lang.\*;**

**Q5)** Qu'affiche le programme ci-dessous?

*public class Maman {*

*public String getVal () {return "Maman";}*

*public Maman () { System.out.print (this.getVal()); }*

*}*

*public class Fifille extends Maman {*

*public String getVal () {return "Fifille";}*

*public Fifille () { super(); }*

*}*

*public class Famille {*

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

*new Fifille();*

*new Maman();*

*}*

*}*

1. **MamanMaman b) MamanFifille c) FifilleFifille d) FifilleMaman**

**Q6)** Quels types de variable sont automatiquement initialisés à leur valeur par défaut?

1. **les attributs de classe et d’instance**
2. **les variables locales**
3. **les paramètres de méthode**
4. **aucune des réponses ci-dessus**

#### Q7) Comment compléter le code ci-dessous pour qu'il affiche la chaîne de caractères "constructeur de base" ?

*class Base {  
 Base (int i){  
 System.out.println("constructeur de base");  
 }  
 Base(){}  
}  
public class Sub extends Base {  
 public static void main (String argv[]){  
 Sub s = new Sub();  
 /\*Un\*/  
 }*

*Sub()  
 {  
 /\*Deux\*/  
 }  
 public void sub()  
 {  
 /\*Trois\*/  
 }  
}*

1. **Remplacer /\*Un\*/ par Base(10);**
2. **Remplacer /\*Un\*/ par super(10);**
3. **Remplacer /\*Deux\*/ par super(10);**
4. **Remplacer /\*Trois\*/ par super(10);**

**Q8)** Identifiez les instructions qui déclarent et instancient correctement des tableaux de 4 entiers.

**a) int[] n = new int[4];**

**b) int[] n = { 3, 4, 2, 30 };**

**c) int n[] = new int[3];**

**d) int n[4];**

**Q9)** Dans le code ci-dessous, choisissez la définition la plus vraisemblable de la méthode ioAppel :

*import java.io.\*;*

*public class Test {*

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

*Test t= new Test();*

*t.uneMethode();*

*}*

*void uneMethode () {*

*try {*

*ioAppel();*

*} catch (IOException e) {}*

*}*

*}*

**a) public void ioAppel() throws IOException {...}**

**b) public void ioAppel() throw IOException {...}**

**c) public void ioAppel() {...}**

**d) public void ioAppel throws IOException () {...}**

**Q10)** Quelle classe n’a pas de classe mère ?

**(a)** Orpheline

**(b)** String

**(c)** Object

**(d)** une classe abstraite

**c)** La classe Object est l’ancêtre de toutes les autres classes.

**Q11)** Qu’est-ce qui est faux pour une interface ?

**(a)** peut être le type d’une référence

**(b)** déclare des méthodes sans les implémenter

**(c)** peut être implémentée

**(d)** peut être instanciée

**d)** L’instanciation est impossible pour une interface, tout le reste est autorisé.

**Q12)** Pour les classes A et D définies comme suit :

*public class A { public class D extends A {*

*public static int f(int x) {return(x+5) ; } public static int f(int x) {return(x+4) ; }*

*public int g(int x) {return (3) ; } public int g( int x) {return (x+8) ; }*

*} }*

Qu’affichera le code suivant ?

*D d=new D() ; A a =d ;*

*System.out.println(a.f(2)\*a.g(3)) ;*

**(a)** 18

**(b)** 21

**(c)** 66

**(d)** 77

**d)** La méthode f() est statique, la variante utilisées est déterminée par la classe de la référence x (c’est à dire A), donc a.f(2)= 2+5. La méthode g() est dynamique, la variante utilisée est déterminée par la vraie classe de l’objet référé par x (dans notre cas D), donc a.g(3)=3+8. D’où la réponse 77.

**Q13)** Trouver la phrase qui n’est pas une caractérisation correcte de polymorphisme :

**(a)** est un mécanisme consistant à rassembler les données et les méthodes au sein d’une structure en cachant l’implémentation de l’objet.

**(b)** signifie que la même opération peut se comporter différemment sur différentes classes de la hiérarchie.

**(c)** offre la possibilité à plusieurs objets de natures différentes d’exposer une interface identique au système et ainsi répondre à un même message d’une manière qui leur est propre.

**(d)** consiste à autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

**a)** La première phrase est en fait une définition de l’encapsulation, les trois autres décrivent divers aspects du polymorphisme

**Q14)** Etant donné que la classe Triangle étend la classe Figure, trouvez la ligne correcte parmi les suivantes

**(a)** Triangle x= new Triangle() ; Object y = (Object) x ; Triangle z=y ;

**(b)** Figure y =new Figure() ; Triangle x= (Triangle) y ; Figure z=x ;

**(c)** Triangle x= new Triangle() ; Figure y = x ; Triangle z=(Triangle) y ;

**(d)** Figure y =new Figure() ; Triangle x= (Triangle) y ; Figure z=(Figure) x ;

**c)** est la bonne réponse :

1. KO La troisième instruction est un downcasting implicite qui mène à une erreur de compilation.
2. KO La compilation se fait sans erreurs, mais à l’exécution le downcasting explicite de la deuxième instruction produit une ClassCastException, comme une simple Figure référée par y n’est pas un Triangle.
3. OK Ici tout va bien : l’objet construit par la première instruction est un Triangle, la deuxième instruction est un upcasting implicite, la troisième instruction est un downcasting explicite qui se déroule bien.
4. KO La compilation se fait sans erreurs, mais à l’exécution le downcasting explicite de la deuxième instruction produit une ClassCastException, comme une simple Figure référée par y n’est pas un Triangle.

**Q15)** Pour la classe D définie comme suit :

*public class D {*

*public int x ;*

*public D() {x=3 ; } ;*

*public D( int a){this() ; x=x+a ;}*

*public D( int a, int b){this(b) ; x= x-a ;}*

*}*

Qu’affichera le code suivant ?

*D a=new D(5,6) ;*

*System.out.println(a.x) ;*

**(a)** 1

**(b)** 2

**(c)** 3

**(d)** 4

**d)** Il suffit de comprendre que l’appel de constructeur D(5,6) commence par appeler D(6) qui appelle à son tour D().

**Q16)** On définit la méthode permuter

*public static void permuter (String s1, String s2, int x1, int x2){*

*String tmp1=s1 ; s1=s2 ; s2=tmp1 ;*

*int tmp2=x1 ; x1=x2 ; x2=tmp2 ;*

*}*

On l’applique dans le contexte suivant :

*String a=”bon” ; String b=”jour” ; int c=3 ; int d =4 ;*

*permuter(a, b, c, d) ;*

Quelles seront les valeurs de *a, b, c, d* après l’exécution de ce code ?

**(a)** ”bon”, ”jour”, 3, 4

**(b)** ”jour”, ”bon”, 3, 4

**(c)** ”bon”, ”jour”, 4, 3

**(d)** ”jour”, ”bon”, 4, 3

**a)** Avec le passage de paramètres par valeur cette méthode ne change rien.

**Q17)** Pour la classe définie comme suit :

*public class Bidon {*

*int x ;*

*}*

Lequel des programmes est faux ?

**(a)** Bidon a=new Bidon(3) ;

**(b)** Bidon a=new Bidon() ; String s=a.toString() ;

**(c)** Bidon a=new Bidon() ; boolean b= a.equals(”bonjour”) ;

**(d)** Bidon a=new Bidon() ; boolean b= a.equals(a) ;

1. new Bidon(3) fait appel à un constructeur inexistant. Toutes les autres lignes utilisent des méthodes héritées de la classe Object ce qui est toujours possible.

**Q18)** Pour les classes A et B définies comme suit:

*public class A { public class B extends A {*

*public int x; public B() {x++;}*

*public A() {x=5; } public B(int i){this(); x=x+i; }*

*} public B(String s){super(); x- -; }*

*}*

Qu’affichera le code suivant ?

*B b1=new B(); B b2 =new B(2003); B b3= new B(”Bonjour”);*

*System.out.println(b1.x + ” et ” + b2.x + ” et encore ” + b3.x );*

**(a)** 6 et 2009 et encore 4

**(b)** 1 et 2004 et encore 4

**(c)** 1 et 2004 et encore 2003

**(d)** autre chose

1. Le constructeur B() n’appelle explicitement ni this(), ni super(). Donc, par convention, le constructeur de la super-classe A est appelée (implicitement) avant de procéder. Ceci donne b1.x=6. Le constructeur B(2003) appelle le constructeur précédent avec le this(), ce qui donne b2.x=6. Ensuite on y ajoute 2003, ce qui donne finalement b2.x=2009 Le constructeur B(”Bonjour”) appelle le constructeur de la super-classe A avec le super(). Ceci donne b3.x=5. Ensuite on le décrémente et on a finalement b3.x=4.

**Q19)** Pour la classe C définie comme suit:

*public class C {*

*public static int i;*

*public int j;*

*public C() {i++; j=i; }*

*}*

Qu’affichera le code suivant?

*C x=new C(); C y=new C(); C z= x;*

*System.out.println(z.i + ” et ” + z.j);*

**(a)** 2 et 2

**(b)** 1 et 1

**(c)** 2 et 1

**(d)** 1 et 3

**c)** On remarque d’abord, que i est une variable (statique) de classe commune à toutes les instances, tandis que chaque objet de la classe a son propre j. Donc, après la première affectation on a i=1, x.j=1; après la deuxième: i=2, y.j=2 (x.j a resté inchangé et égal `a 1); la troisième n’appelle pas le constructeur mais fait z référencer le même objet que x. D’où z.i est la valeur globale de i, c-à-d 2, et z.j=x.j=1.

**Q20)** L’interprétation des programmes Java est effectuée par :

(a) API

(b) JDK

(c) JVM

(d) AWT