QCM de Java corrigé

1. Java est un langage

   (a) Compilé

   (b) Interprété

   [(c)] Compilé et interprété

   (d) Ni compilé ni interprété

   Le compilateur compile le code source vers un bytecode, la machine virtuelle Java (JVM) interprète ce bytecode

2. Java est un langage développé par

   (a) Hewlett-Packard

   [(b)] Sun Microsystems

   (c) Microsoft

   (d) Oracle

   Par James Gosling chez Sun.

3. Combien d'instances de la classe A crée le code suivant?

```
A x,u,v;
x=new A();
A y=x;
A z=new A();
```

   (a) Aucune

   (b) Cinq

   (c) Trois

   [(d)] Deux

   Il y a deux instances de A créées par les deux new*, la première est référencée par x et y, la deuxième par z.*

4. Pour la classe B définie comme suit:

```
class B {
public B(){System.out.print("Ciao");};
public B(int i) {this(); System.out.println("Bonjour "+i);};
}
```

   qu'affichera l'instruction suivante?

```
B monB=new B(2003);
```

   (a) erreur de compilation

   (b) erreur d'exécution

   [(c)] CiaoBonjour 2003

   (d) Bonjour 2003

   *L'instruction invoque le constructeur avec un argument entier (2003). Ce dernier appelle explicitement le constructeur sans arguments (*this()*) qui imprime "Ciao", et ensuite le message "Bonjour 2003" est imprimé.*

5.

[(a)] Une classe peut implémenter plusieurs interfaces mais doit étendre une seule classe

(b) Une classe peut implémenter plusieurs classes mais doit étendre une seule interface

(c) Une classe peut implémenter plusieurs classes et peut étendre plusieurs interfaces

(d) Une classe doit implémenter une seule interface et étendre une seule classe

C'est comme ça

6. La liaison tardive est essentielle pour assurer

(a) l'encapsulation

[(b)] le polymorphisme

(c) l'héritage

(d) la marginalisation

La marginalisation n'a rien à voir avec la programmation. La liaison tardive dynamique permet d'utiliser pour chaque objet sa propre version d'une méthode (en fonction de la classe de l'objet déterminée à l'exécution). Ceci permet d'obtenir un code polymorphe.

7. Étant donné que la classe Grande étend la classe Petite, trouvez une ligne correcte parmi les suivantes

(a) `Petite y =new Petite(); Grande x= (Grande)y; Petite z=x;`

La deuxième affectation `Grande x= (Grande)y;` *essaye de transformer un objet (référencé par y) de la classe* `Petite` *vers un objet de sa sous-classe* `Grande`*. Un tel downcasting est impossible.*

[(b)] `Grande x= new Grande(); Petite y = x; Grande z=(Grande)y;`

Tout va bien. On crée un objet de classe `Grande` *référencé par x. Ensuite on fait une variable y (de type* `Petite`*) référencer le même objet – c'est un upcasting explicite qui est toujours possible. À la fin on fait encore une référence z (cette fois* `Grande`*) sur ce même objet. Ce dernier downcasting est possible parce que l'objet est en fait une instance de la classe* `Grande`*.*

(c) `Grande x= new Grande(); Petite y = x; Grande z=y;`

C'est presque comme dans le cas précédent, mais la dernière affectation `Grande z=(Grande)y;` *est un downcasting implicite, ce qui est interdit.*

(d) `Petite y =new Petite(); Grande x= (Grande)y; Petite z=(Petite)x;`

`Grande x=(Grande)y;` *est un downcasting impossible, comme dans le (a).*

8. Pour la classe C définie comme suit:

```
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

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 à 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.

9. Pour les classes A et B définies comme suit:

```
class A {
public int x;
public A() {x=5; }
}
```

```
class B extends A {
public B() {x++;}
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

Le constructeur B() n'appelle explicitement ni this(), ni super(). Donc, par convention, le constructeur de la super-classe A est appelé (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 decrémente et on a finalement b3.x=4.

10. Pour les classes Machin et Bidul définies comme suit:

```
class Machin {
public int f() {return(5) };
public static int g() {return (6);}
}
```

```
class Bidul extends Machin {
public int f() {return(2) };
public static int g() {return (4); }
}
```

qu'affichera le code suivant?

```
Bidul b=new Bidul(); Machin m =b;
System.out.println(m.f()*m.g());
```

(a) 30

(b) 20

(c) 8

(d) 12

La methode g est statique, la version utilisée est déterminée par le compilateur en fonction du type de la référence. Comme m est une référence Machin*, m.g() - c'est toujours la méthode g de la classe* Machin *(valeur 6). Pour f tout est différent: sa version utilisée est déterminée dynamiquement (à l'execution)par la JVM en fonction du type de l'objet. Dans le cas de m.f(), m fait référence à une instance de* Bidul*, c'est donc la version de f redéfinie dans la classe* Bidul *qui est invoquée (valeur 2). D'où la réponse.*