## Cuestiones UT7 (III) – Herencia, polimorfismo e interfaces.

41) Define los términos *early binding* (*static binding*) y *late binding* (*dynamic binding*). Qué métodos se asocian en Java mediante *static binding*? Y mediante *dynamic binding?*

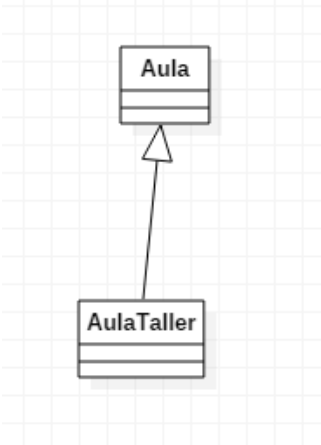
**Early binding hace referencia a tiempo en compilación. Y dynamic a tiempo en ejecución.**

**Los métodos asociados a static binding son los que se declaran como static, final y private.**

**Y los dynamic binding son todos los demás.**

42) Sea:

```
AulaTaller at = new AulaTaller();
Aula a = at;    (a)
AulaTaller at2 = (AulaTaller) a;  (b)
```




Son correctas las asignaciones (a) y (b) ?

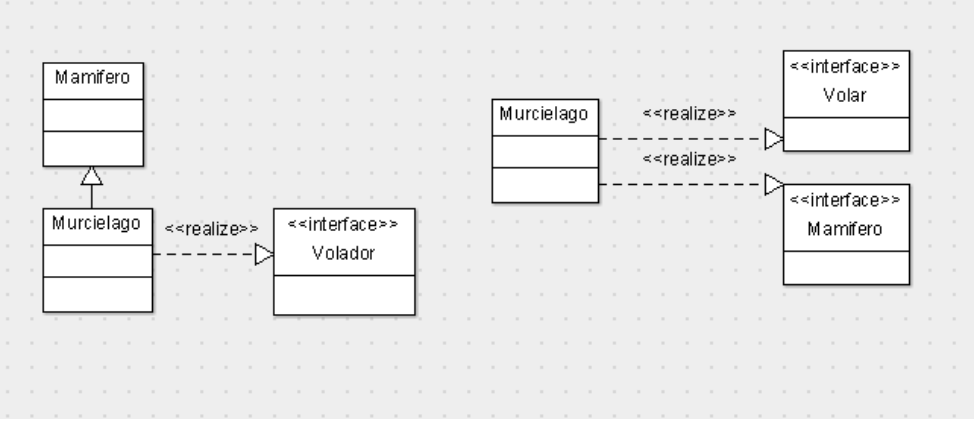
**Son correctas ambas asignaciones.**

Qué se está haciendo en (a) y (b): downcasting / upcasting?

**En a se está haciendo upcasting y en b downcasting.**

43) Es la definición de un tipo sin implementación, representa un protocolo, un contrato que una clase que lo implementa ha de cumplir. **Hablamos de una interfaz.**

44) Un murciélago es un mamífero volador. Dibuja un diagrama UML que representa esta relación en el que aparezca la clase Murcielago en una relación de herencia e implementado un interfaz. Haz lo mismo con dos interfaces.



45) Qué debe hacer una clase para poder implementar un interfaz (hasta Java 8)?

**Dar código a todos los métodos contenidos en la interfaz.**

46) ¿Cómo son los métodos en un interfaz? **abstractos**
¿Y los atributos? **Una interfaz no tiene atributos, solo constantes y métodos abstractos**.

47)
Indica Verdadero / Falso.

| | |
|---|---|
| a) Los constructores se heredan | f |
| b) Las clases abstractas pueden utilizarse como tipos | v |
| c) Las clases abstractas pueden instanciarse | f |
| d) Los interfaces no pueden instanciarse | v |
| e) Una clase solo puede implementar un interfaz | f |
| f) Una clase puede heredar de varias clases | f |
| g) Los interfaces pueden utilizarse como tipos | v |
| h) Una variable cuyo tipo es un interfaz no puede almacenar objetos de clases que implementan el interfaz o subclases de una clase que implemente el interfaz. | f |
| i) Un método *final* puede redefinirse | f |
| j) Una clase abstracta no puede tener constructores | f |
| k) Para poder redefinir un método en una subclase ha de ser abstracto en la superclase | f |
| l) Object es una clase abstracta | f |
| m) Todos los métodos de una clase abstracta son abstractos | f |
| n) La llamada a *super*() en un constructor puede hacerse en cualquier momento | f |
| o) Una superclase es más específica que una subclase que es más general | f |
| p) Las subclases responden a los mensajes definidos en las superclases | v |
| q) Solo las clases abstractas pueden tener métodos abstractos | v |
| r) Una clase abstracta solo puede tener métodos abstractos | f |
| s) Una clase abstracta puede no tener métodos abstractos | v |
| t) Una clase concreta puede tener métodos abstractos | f |
| u) Los métodos abstractos se implementan en la clase en la que se definen | f |
| v) Un método abstracto puede ser final | f |
| w) Un interfaz puede contener atributos no static | f |

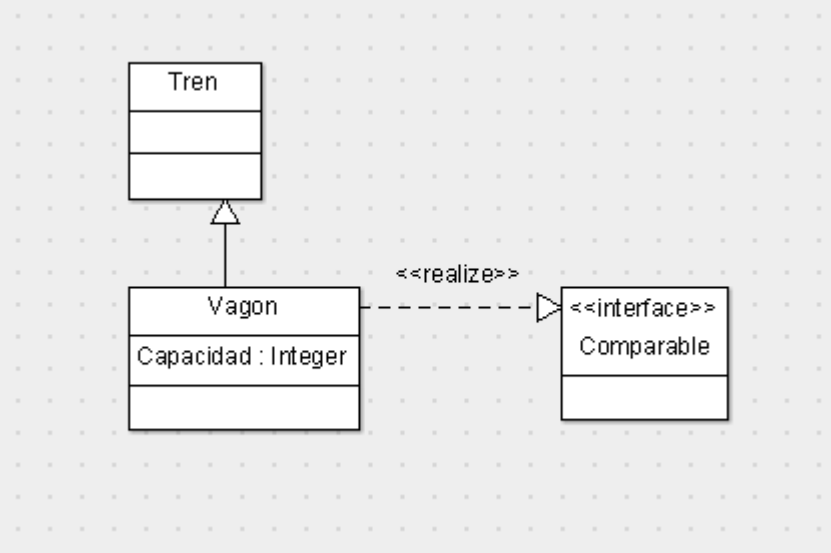
| | |
|---|---|
| x) Un interfaz en Java 8 puede contener métodos por defecto | v |
| y) Un interfaz en Java 8 no puede contener métodos static | f |

48) ¿Es correcto? Razona la respuesta.

```
List<String> lista = new ArrayList<String>();
Map<String, Persona> lista = new TreeMap<String, Persona>();
```

**Si, es correcto por que porque un objeto que implemente una interfaz puede ser almacenado en una variable del tipo de la interfaz.**

49) La clase Vagón tiene entre otros, el atributo *capacidad* de tipo entero. La clase Tren que contiene una serie de vagones ha de poder ordenarse según la capacidad de los vagones que tiene. ¿Qué harás? Dibuja el diagrama UML que muestre la respuesta.



50) Una clase Aula queda definida por su nombre y su superficie en metros cuadrados. Define la clase (sus atributos) e incluye en ella lo necesario para poder determinar si dos aulas son iguales y, para poder ordenar las aulas. Dos aulas son iguales si su nombre y superficie coinciden. La relación de orden natural entre dos aulas se establece en relación a su superficie.

```
public class Aula implements Comparable<Aula> {

	private int superficie;
	private String nombre;

	public int getSuperficie() {

		return superficie;

	}

	@Override
	public int compareTo(Aula aux) {

		return (int) Math.signum(this.superficie -
aux.getSuperficie());
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result
				+ ((nombre == null) ? 0 : nombre.hashCode());
		result = prime * result + superficie;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Aula other = (Aula) obj;
		if (nombre == null) {
			if (other.nombre != null)
```

```
                return false;
            } else if (!nombre.equals(other.nombre))
                return false;
            if (superficie != other.superficie)
                return false;
            return true;
        }

}
```

51) En el ejemplo anterior indica cómo ordenar dos aulas si no incluimos el método compareTo() (es decir si no utilizamos Comparable)

**Con collections.sort(listaAulas, Comparator());**

52) ¿Qué hace el método getClass ?

**Devuelve un objeto que indica a que clase pertenece.**
¿En qué clase está?

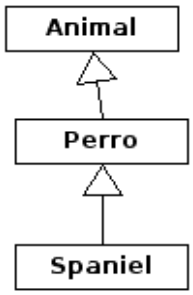
**En Object.**

La signatura de este método dice: public final Class getClass()

¿Puedo redefinir este método en mi clase puesto que lo heredo?

**No, ya que incluye la palabra final.**

53) Diferencia entre instanceof y getClass(). Analízalo con el siguiente ejemplo:




```
Animal a = new Perro();
Animal miPerro  = new Spaniel();
Perro p;

        if (miPerro  instanceof Perro)
                p = (Perro) miPerro;

        if (miPerro.getClass() .equals(Perro.class))
                p = (Perro) miPerro;
```

**La primera sentencia dara true ya que analiza esa clase y todas sus hijas, por el contrario la segunda dara false ya que no es de la clase concreta de perro sino de Spaniel.**

54) ¿Qué requisito debe cumplir una clase para poder clonarla con el método clone()?

**Se debe implementar la interfaz clonable a las clases que se deseen clonar.**

55) Puede aparece un interfaz como tipo de un argumento o valor de retorno en un método? **Si**

56) Qué característica tiene el interface Cloneable?

**Que los objetos que la implementen pueden ser clonados.**

57) Puede un interface contener solo constantes? **Si**
Indica un interfaz de la API de Java que lo haga.

**Cloneable**

58) Sean las siguientes clases CD y Cancion.

```
public class  CD
{
    private  ArrayList<Cancion>
canciones;

      public void addCancion(Cancion
c)
    {
          if (!canciones.contains(c))
          {
                canciones.add(c);
          }
    }

     }
```

```
public class  Cancion
{
    private  String titulo;
    private int duracion;

                ….....

     }
```

A pesar de que el método addCancion() comprueba si ya existe una canción antes de añadirla se están admitiendo canciones duplicadas en la colección. ¿Qué puede estar ocurriendo?

**Ya que contains se apoya en equals para funcionar necesitaremos redefinir el metoido equals para que contains sepa lo que tiene que comparar para realizarlo bien.**

59) ¿Puede un interface extender (heredar) de otro interfaz? Antes de responder busca en la API el interface Collection y el interfaz Set.

**Si**

Anota la cabecera de definición del interface Set.

```
public interface Set<E> extends Collection <E>
```

60) Supongamos el siguiente escenario

```
interface A
{
    int   hacerUno();
    int    hacerDos();

 }

interface B extends A
{
    int  hacerTres();


 }
```

```
public class  Prueba implements  A
{


 }

public class  Demo implements  B
{


 }
```

¿Qué métodos debe implementar la clase Prueba?

**HacerUno y hacerDos;**

¿Y la clase Demo?

**HacerUno, HacerDos y hacerTres;**

61) Sean las siguientes clases CD y Cancion. Asumimos en la clase Cancion los accesores y mutadores ya definidos.

```
 public class  CD
{
        private  TreeSet<Cancion>
        canciones;

public void addCancion
(Cancion c)
        {
                canciones.add(c);
        }

        }

 }
```

```
public class  Cancion
 {
        private  String titulo;
        private int duracion;

            ……..........

}
```

Los métodos equals() y toString() se han redefinido adecuadamente en la clase Cancion.
Sin embargo al añadir una canción al obtenemos un error de ejecución. ¿Por qué ocurre esto? ¿Cómo lo corregimos?

**Dado que la estructura treeSet almacena los objetos de forma ordenada los objetos Canción deben saber cómo ordenarse y para ello tienen que implementar la interfaz Comparable y redefinir el método compareTo();**

62. Para que la colección HashMap<Estudiante, Integer> que asocia objetos Estudiante con sus notas correspondientes funcione correctamente, ¿qué métodos debe redefinir la clase Estudiante?

**El equals y el hasCode()**

63. Dada las siguientes definiciones:

```
public class  Coche
{
        private  int velocidad;

           public Coche(int
velocidad)
        {
                this.velocidad =
velocidad;

        }
          ……….

}
```

```
Coche[] array = new Coche[3];
array[0] = new Coche(120);
array[1] = new Coche(200);
array[2] = new Coche(230);

 Coche objetivo = new Coche(120);
System.out.println(Arrays.binarySearch(array,
jetivo);
```

Al ejecutar el método binarySearch() para buscar el coche *objetivo* tenemos un error de ejecución. ¿qué puede estar pasando?

**Para hacer una búsqueda binaria el array debe estar ordenado sino dara error. Para poder hacer esto la clase coche deberá implementar la interfaz Comparable y redefinir el compareTo();**

64. Dada la definición List<String> palabras = new ArrayList<String>(); y asumiendo que se han añadido una serie de palabras:

a) Ordena la colección en orden creciente según su orden natural con un método de la clase Collections

   **Collections.sort(palabras);**

b) Ordena la colección en orden creciente según su orden natural con un método de la clase List. Qué característica tiene ese método?

   **Palabras.sort(Comparator<>) // hay que pasarle un comparator para que sepa como ordenar;**

c) Ordena la colección en orden decreciente según marca su orden natural con un método de la clase Collections

**Collections.sort(palabras, Collections.reverseOrder());**

d) Ordena la colección en orden decreciente según marca su orden natural con un método de la clase List

**Palabras.sort(Collections.reverseOrder());**