INSTRUCCIONES

Lee detenidamente las siguientes instrucciones, antes de abordar la realización de la prueba, con el objetivo de tener claro los aspectos a valorar y/o lo demandado en la misma.

1. Puede haber preguntas con más de una posible respuesta.
2. Indica la solución que creas correcta de cada pregunta en la tabla del final.
3. Por supuesto PREGUNTA cualquier duda o aclaración que necesites.
4. TIEMPO ESTIMADO: 60 min

A rellenar por el alumno:

Nombre:

Apellidos:

Fecha:

A. Preguntas tipo test (elige la opción correcta)

1. Sobre **tipos primitivos vs referencia**, indica lo correcto:
   A) Los primitivos siempre son 0.
   B) Los de referencia pueden ser null; los primitivos no.
   C) Los de referencia tienen tamaño fijo; los primitivos no.
   D) Ambos pueden ser null.

B. Los tipos de referencia pueden ser null; los primitivos nunca. (Por eso Integer x=null es válido y int x=null no).

2. ¿Cuándo es más apropiado switch frente a if-else?
   A) Cuando cada condición evalúa expresiones complejas no relacionadas.
   B) Cuando todos los valores dependen de la misma variable.
   C) Cuando hay que evaluar rangos numéricos.
   D) Nunca, if-else es más legible.

B. switch encaja cuando las ramas dependen de la misma variable/expresión con valores concretos.

3. Sobre **constructores** y encadenamiento:
   A) this(...) debe ser la última instrucción del constructor.
   B) super(...) y this(...) pueden llamarse en el mismo constructor.
   C) Si no llamas a super(...), el compilador inserta super() implícito.
   D) Un constructor puede devolver void.

C. Si no invocas super(...), el compilador inserta super() implícito.
this(...)/super(...) deben ir en primera línea y no pueden coexistir.

4. La palabra clave **new**:
   A) Reserva memoria, invoca el constructor y devuelve una **referencia**.
   B) Crea siempre objetos inmutables.
   C) Devuelve una copia por valor del objeto.
   D) Sólo puede usarse con tipos primitivos.

A. new reserva memoria, llama al constructor y devuelve la referencia del objeto creado.

5. **Métodos estáticos vs de instancia**:
   A) Un estático puede acceder directamente a campos de instancia.
   B) Un método de instancia puede acceder a miembros estáticos.
   C) Los estáticos pueden ser sobrescritos.
   D) Los de instancia pertenecen a la clase, no al objeto.

B. Un método de instancia sí puede acceder a miembros estáticos; al revés (estático → instancia) no, sin objeto.

6. **Sobrecarga** y **sobrescritura**:
   A) Sobrecargar exige mismo nombre y mismos parámetros.
   B) Sobrescribir admite cambiar el tipo de retorno covariante.
   C) Sobrescribir permite reducir la visibilidad (p. ej., de public a private).
   D) Sobrecargar obliga a usar @Override.

B. En sobrescritura se permite retorno covariante; no puedes reducir visibilidad y la sobrecarga cambia parámetros.

7. En Java, la **herencia múltiple** de implementación:
   A) Se permite con extends varias clases.
   B) No existe; se logra vía múltiples interfaces.
   C) No existe bajo ninguna forma.
   D) Se permite sólo con final.

B. No hay herencia múltiple de implementación; se simula con múltiples interfaces.

8. Dado String s = "Hola mundo"; ¿qué es cierto?
   A) s.replace("o","O") modifica el objeto directamente.
   B) s.substring(0,4) devuelve "Hola".
   C) s.indexOf("z") lanza excepción.
   D) s.toUpperCase() siempre cambia s.

B. substring(0,4) devuelve "Hola". String es inmutable: replace/toUpperCase crean nuevos objetos; indexOf devuelve -1, no lanza excepción.

9. Paso de parámetros en Java:
   A) Siempre por referencia.
   B) Siempre por valor (copia del **valor**).

C) Primitivos por valor, objetos por referencia.
D) Depende de la JVM.

C. Cuando se pasa un primitivo se pasa una copia del valor si se pasa un objeto es la referencia al objeto lo que se pasa.

10. **Métodos abstractos**:
    A) Deben declararse en clases abstractas o interfaces.
    B) Pueden tener cuerpo vacío {}.
    C) Pueden ser private.
    D) Pueden ser static.

A. Un método abstracto solo en clases abstractas o interfaces; no tiene cuerpo, ni puede ser private o static.

11. **Clases abstractas**:
    A) No pueden tener métodos concretos (con código).
    B) Pueden tener constructores.
    C) Se pueden instanciar directamente.
    D) Deben marcarse final.

B. Las clases abstractas pueden tener constructores y métodos concretos; lo que no pueden es instanciarse.

12. **¿Cuál es la diferencia principal entre una interfaz y una clase abstracta en Java?**
    A) Una interfaz puede tener métodos concretos y abstractos, mientras que una clase abstracta solo puede tener métodos abstractos.
    B) Una interfaz no puede tener variables de instancia, mientras que una clase abstracta puede tener variables de instancia.
    C) Una interfaz puede ser instanciada directamente, mientras que una clase abstracta no puede ser instanciada directamente.
    D) Una clase abstracta puede ser implementada por múltiples clases, mientras que una interfaz solo puede ser implementada por una clase.

B. Las interfaces no tienen variables de instancia (sus campos son public static final); una clase abstracta sí puede tener estado.

13. Sobre **interfaces**:
    A) Una clase sólo puede implementar una interfaz.
    B) Una interfaz puede extender varias interfaces.
    C) Los campos de una interfaz no son final.
    D) Los métodos de interfaz son private por defecto.

B. Una interfaz puede extender varias interfaces. (Una clase puede implementar varias interfaces).

14. **Encapsulación** con getters/setters:
    A) Permite validar y proteger la invariancia antes de asignar.
    B) Impide cualquier cambio de estado.
    C) Es equivalente a usar campos public.
    D) No aplica a clases inmutables.

A. Getters/setters permiten validar y preservar invariantes antes de asignar; no equivalen a campos public.

15. **¿Cuál es la diferencia principal entre las interfaces List y Set en Java?**
    A) List permite elementos duplicados y mantiene el orden de inserción, mientras que Set no permite elementos duplicados y no garantiza el orden de inserción.
    B) List no permite elementos duplicados y no mantiene el orden de inserción, mientras que Set permite elementos duplicados y mantiene el orden de inserción.
    C) List y Set permiten elementos duplicados, pero solo List mantiene el orden de inserción.
    D) List y Set no permiten elementos duplicados, pero solo Set mantiene el orden de inserción.

A. List permite duplicados y mantiene orden de inserción; Set no permite duplicados y no garantiza orden (salvo implementaciones concretas).

Pregunta 1:

Considerando el siguiente código, ¿cuál será la salida y por qué?

```
class Animal {
  void comer() {
    System.out.println("Animal comiendo");
```

```
  }
}
class AnimalCarnivoro {
  void comer () {
    System.out.println("Animal comiendo carne");
  }
  void cazar() {
    System.out.println("Animal carnívoro cazando");
  }

}
class Lobo extends AnimalCarnivoro {
  void comer () {
    System.out.println("Animal comiendo carne en manada");
  }

}
public class Test {
  public static void main(String[] args) {
    Animal a = new Lobo ();
    if (a instanceof AnimalCarnivoro) {
      Lobo l = (Lobo) a;
      l.comer();
      l.cazar();
    } else {
      a.comer();
    }
  }
```

}

A) Error de compilación en l.cazar()
B) Animal comiendo carne en manada
Animal carnívoro cazando

C) se produce la excepción ClassCastException en la línea Lobo l = (Lobo) a;

D) Animal comiendo carne
Animal carnívoro cazando

Respuesta correcta: B

Explicación: En este código, Animal a = new Dog(); crea una instancia de Dog pero la referencia es de tipo Animal. La condición if (a instanceof Dog) verifica si a es una instancia de Dog (o de una subclase de Dog), lo cual es cierto. Por lo tanto, se ejecuta el bloque dentro del if, donde Dog d = (Dog) a; convierte la referencia a a una referencia de tipo Dog. Luego, d.makeSound() llama al método makeSound de la clase Dog, imprimiendo "Bark", y d.fetch() llama al método fetch de la clase Dog, imprimiendo "Dog fetches the ball".

Pregunta 2:

Considere el siguiente código. ¿Cuál será la salida de la ejecución del código?

```
public class Test {
  public static void main(String[] args) {
    int a = 5;
    modifyPrimitive(a);
    System.out.println("a = " + a);
    MyObject obj = new MyObject();
    obj.value = 5;
    modifyObject(obj);
    System.out.println("obj.value = " + obj.value);
  }
  public static void modifyPrimitive(int x) {
```

```
    x = 10;
  }
  public static void modifyObject(MyObject o) {
    o.value = 10;
  }
}

class MyObject {
  int value;
}
```

A) a = 5
obj.value = 5

B) a = 5
obj.value = 10

C) a = 10
obj.value = 5

D) a = 10
obj.value = 10

Respuesta correcta: B

Explicación: En el método modifyPrimitive, el parámetro x es una copia del valor de a. Cambiar el valor de x dentro del método no afecta al valor original de a, por lo que a sigue siendo 5 después de la llamada al método.

En el método modifyObject, el parámetro o es una copia de la referencia obj. Aunque la referencia es una copia, ambas referencias apuntan al mismo objeto en memoria. Por lo tanto, cuando se modifica o.value dentro del método, se está modificando el mismo objeto al que apunta obj. Como resultado, obj.value es 10 después de la llamada al método.

Pregunta 3:

Qué tres palabras reservadas irían en los puntos donde aparece _____ (por orden de aparición)

```
_______ class Animal {
   abstract void makeSound();
   void sleep() {
      System.out.println("Sleeping...");
   }
}
class Dog _________ Animal {
   void makeSound() {
      System.out.println("Bark");
   }
}
public class Test {
   public static _________ main(String[] args) {
      Animal myDog = new Dog();
      myDog.makeSound();
      myDog.sleep();
   }
}
```

A) public, implements, String

B) interface, implements, void

C) abstract, extends, void

D) *vacio, *vacio, String

*vacio=no poner nada

La correcta es C) abstract, extends, void.

Por qué:

- **_____ class Animal { ... abstract void makeSound(); ... } → Una clase que declara un método abstracto debe marcarse como abstract.**
- **class Dog _____ Animal → Una clase concreta que hereda de una clase (no interfaz) usa extends.**
- **public static _____ main(String[] args) → El main estándar no retorna nada, por eso va void.**

Por qué no las otras:

- A) public class Animal con un método abstracto no compila sin abstract. Además, Dog implements Animal sería para interfaces, no para clases. Y main no retorna String.
- B) Si Animal fuera interface, entonces sleep() debería ser default void sleep(), no simplemente void sleep().
- D) Dejar vacío en Animal no permite tener un método abstracto; también faltaría extends/implements en Dog, y main no puede retornar String.

Pregunta 4:

Qué tres palabras reservadas irían en los puntos donde aparece _____ (por orden de aparición)

```
interface Animal {
  void _____();

  default void sleep() {
    System.out.println("Sleeping...");
  }
}

_____ Dog implements Animal {
  public void makeSound() {
    System.out.println("Bark");
  }
```

```
  public void eat() {
    System.out.println("eating");
  }

}

public class Test {
  public static void main(String[] args) {
    Animal myDog = new _______();
    myDog.makeSound();
    myDog.sleep();
  }
}
```

A) makeSound, class, Dog()

B) makeSound, class, Animal()

C) eat, class, Dog()

D) eat, public, Animal()

La correcta es **A) makeSound, class, Dog()**.

Por qué:

- En interface Animal { void _____(); … } debe ir el método abstracto que la clase implementará: **makeSound** (porque en Dog existe public void makeSound()).
- En _____ Dog implements Animal { … } la palabra reservada que define una clase es **class**.
- En Animal myDog = new _______(); hay que instanciar la clase concreta **Dog**, así que va **Dog()** tras new.

Por qué no las otras:

- **B)** Instancia Animal() (no se puede instanciar una interfaz).

- **C)** Pone eat en la interfaz, pero eat es un método adicional de Dog, no el obligado por la interfaz en este enunciado.
- **D)** Usa public donde debe ir class, y vuelve a intentar instanciar la interfaz Animal().

Pregunta 5:

Considere el siguiente código. **¿Cuál será la salida de la ejecución del código?**

```
import java.util.HashSet;
import java.util.Objects;
import java.util.Set;

class Persona {
  private String dni;
  private String nombre;
  private int edad;

  public Persona(String dni, String nombre, int edad) {
    this.dni = dni;
    this.nombre = nombre;
    this.edad = edad;
  }

  // Getters
  public String getDni() {
    return dni;
  }

  public String getNombre() {
```

```
    return nombre;
  }

  public int getEdad() {
    return edad;
  }
}
public class Test {
  public static void main(String[] args) {
    Set<Persona> personas = new HashSet<>();
    // Añadir personas al Set
Persona p = new Persona("12345678A", "Alice", 30);
    personas.add(p);
    personas.add(new Persona("87654321B", "Bob", 25));
    personas.add(new Persona("12345678A", "Carlos", 40)); // DNI duplicado
        personas.add(p);
    // Imprimir el contenido del Set
    for (Persona persona : personas) {
      System.out.println(persona.getDni());
    }
  }
}
```

A) 12345678A, 87654321B, 12345678A

B) 12345678, 87654321B

C) 12345678, 87654321B, 12345678A, null

D) Hay un error de compilación en la línea:

Persona p = new Persona("12345678A", "Alice", 30);

Pregunta	Respuesta	Pregunta	Respuesta

Pregunta 1		Pregunta 11	
Pregunta 2		Pregunta 12	
Pregunta 3		Pregunta 13	
Pregunta 4		Pregunta 14	
Pregunta 5		Pregunta 15	
Pregunta 6		Pregunta 16	
Pregunta 7		Pregunta 17	
Pregunta 8		Pregunta 18	
Pregunta 9		Pregunta 19	
Pregunta 10		Pregunta 20	