Ejercicios propuestos

▼ Ejercicio 1 - Polimorfismo y Excepciones

Considera el siguiente bloque de código:

```
class Animal {
    void makeSound() throws Exception {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void makeSound() throws RuntimeException {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        try {
            myDog.makeSound();
        } catch (Exception e) {
            System.out.println("Exception caught");
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Dog barks
2- Animal makes a sound
3- Exception caught
4- Compilation error

Análisis del código

1. La clase `Dog` hereda de la clase `Animal` , y se está realizando Override del método makeSound.
2. Override de métodos permite que el submétodo arroje una subclase de la excepción arrojada en el método padre.
3. Se instancia un perro en una variable de tipo `Animal` .
4. Se invoca al método `makeSound()` desde el objeto creado y dentro de un bloque try-catch para manejar la excepción.
5. El bloque es capaz de manejar la excepción.
6. El método que se ejecuta corresponde al de la clase `Dog` , pues ese es el tipo de objeto que tenemos.

Respuesta

1- Dog barks

▼ Ejercicio 2 - Hilos (Threads)

Considera el siguiente bloque de código:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Thread is running (impreso una vez)
2- Thread is running (impreso dos veces)
3- Thread is running (impreso dos veces, en orden aleatorio)
4- Compilation error

Análisis del código

1. Se crea una clase que hereda de `Thread` y se hace Override del método `run()` para que al ejecutar nuestro hilo se imprima `Thread is running` .
2. En el método `main()` se instancian dos de nuestros hilos personalizados.
3. Se invoca el método `start()` de ambos hilos.
4. Se imprime `Thread is running` dos veces y el orden de impresión depende de la JVM.

Respuesta

3- Thread is running (impreso dos veces, en orden aleatorio)

▼ Ejercicio 3 - Listas y Excepciones

Considera el siguiente bloque de código:

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        try {
            for (int i = 0; i <= numbers.size(); i++) {
                System.out.println(numbers.get(i));
```

```
            }
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Exception caught");
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- 1 2 3 Exception caught
2- 1 2 3
3- Exception caught
4- 1 2 3 4

Análisis de código

1. En el método `main()` se instancia un `ArrayList` que almacena objetos tipo `Integer` .
2. A la lista se agregan elementos a través del método `add()` .
3. Los valores ingresados se envuelven en su wrapper automáticamente.
4. Se utiliza un bloque try-catch para prevenir que la iteración fuera de rango de la lista detenga la ejecución.
5. El ciclo de iteración rebasa los índices del arreglo. El arreglo cuenta con 3 elementos, mientras que el ciclo for realizará 4 iteraciones.
6. Se completan la iteraciones que si corresponde a los índices adecuadamente y posteriormente se atrapa la excepción y se imprime `"Exception caught"` .

Respuesta

1- 1 2 3 Exception caught

▼ Ejercicio 4 - Herencia, Clases Abstractas e Interfaces

Considera el siguiente bloque de código:

```
interface Movable {
    void move();
}

abstract class Vehicle {
    abstract void fuel();
}

class Car extends Vehicle implements Movable {
    void fuel() {
        System.out.println("Car is refueled");
    }

    public void move() {
        System.out.println("Car is moving");
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        Vehicle myCar = new Car();
        myCar.fuel();
        ((Movable) myCar).move();
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Car is refueled Car is moving
2- Car is refueled
3- Compilation error
4- Runtime exception

Análisis de código

1. Se tiene una clase `Car` que hereda de la clase abstracta `Vehicle` e implementa la interfaz `Movable` .
2. La implementación de los métodos se realiza de manera adecuada, no se altera la visibilidad.
3. En el método `main()` se instancia un carro en una variable de tipo `Vehicle` .
4. Se invoca al método `fuel()` , el método imprime `"Car is refueled"` debido a que nos fijamos en el tipo de objeto, además, es el único método definido.
5. Se realiza el cast de la variable de `Vehicle` a `Movable` y se ejecuta el método `move()` .
6. El resultado se imprime

Respuesta

1- Car is refueled Car is moving

▼ Ejercicio 5 - Polimorfismo y Sobrecarga de Métodos

```
class Parent {
    void display(int num) {
        System.out.println("Parent: " + num);
    }

    void display(String msg) {
        System.out.println("Parent: " + msg);
    }
}

class Child extends Parent {
    void display(int num) {
        System.out.println("Child: " + num);
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(5);
        obj.display("Hello");
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Child: 5 Parent: Hello
2- Parent: 5 Parent: Hello
3- Child: 5 Child: Hello
4- Compilation error

Análisis de código

1. La clase `Parent` tiene sobrecargado el método display.
2. La clase `Child` hereda de la clase `Parent` y aplica Override al método `display(int num)` .
3. En el método `main()` se crea un objeto de tipo `Child` en una variable de referencia `Parent` .
4. Se invoca 2 veces al método `display` , primero con un `int` y posteriormente con un `String` .
5. Nos fijamos en el tipo de objeto, por lo que la invocación de `display(5)` imprime `Child: 5` . La invocación de `display("Hello")` imprime `Parent: Hello` ya que el método se hereda.

Respuesta

1- Child: 5 Parent: Hello

▼ Ejercicio 6 - Hilos y Sincronización

Considera el siguiente bloque de código:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

class MyThread extends Thread {
    private Counter counter;

    public MyThread(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new MyThread(counter);
```

```
        Thread t2 = new MyThread(counter);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(counter.getCount());
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- 2000
2- 1000
3- Variable count is not synchronized
4- Compilation error

Análisis de código

1. La clase `Counter` define un atributo `count=0` , un método `getCount()` que regresa el valor de `count` y el método `increment()` que incrementa en 1 a `count` , y en donde `synchronized` indica que el método únicamente puede ser utilizada por un hilo a la vez.
2. La clase `MyThread` hereda de la clase `Thread` , tiene un `Counter` como atributo y este se asigna a través de su constructor. Además se hace Override del método `run()` y se indica que al ejecutar el hilo se realice un incremento al `Counter` 1000 veces.
3. En el método `main()` se indica que puede ser lanzada `InterruptedException` .
4. Se instancia un contador.
5. Se instancian dos hilos de la clase `MyThread` y se les pasa la referencia del mismo `counter`
6. Se invoca al método `start()` para ambos hilos.
7. Se invoca al método `join()` , de manera que el hilo principal esperará a que los demás hilos se vuelvan a unir al principal.
8. Se imprime 2000 debido que se manejo el mismo contador para ambos hilos, entonces un hilo fue del 1 al 1000, y el segundo de 1001 hasta 2000. Hay que considerar que los incrementos son gestionados por la JVM y no se realizan con un orden específico.

Respuesta

1- 2000

▼ Ejercicio 7 - Listas y Polimorfismo

Considera el siguiente bloque de código:

```
import java.util.ArrayList;
import java.util.List;

class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
```

```
    void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        animals.add(new Dog());
        animals.add(new Cat());
        animals.add(new Animal());

        for (Animal animal : animals) {
            animal.makeSound();
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Animal sound Animal sound Animal sound
2- Bark Meow Animal sound
3- Animal sound Meow Bark
4- Compilation error

Análisis de código

1. Se realiza la importación de `List` y de `ArrayList` desde `java.util` .
2. La clase `Animal` define el método `makeSound()` .
3. La clase `Dog` hereda de la clase `Animal` y hace un Override al método `makeSound()` .
4. La clase `Cat` hereda de la clase `Animal` y hace un Override al método `makeSound()` .
5. En la clase `main()` se crea un `ArrayList` que soporta objetos de tipo `Animal` .
6. Se agregan instancias de `Dog` , `Cat` , `Animal` al arreglo.
7. Se itera sobre el arreglo y por cada elemento se invoca el método `makeSound()` .
8. Debido al polimorfismo se imprime el método respectivo de cada objeto.

Respuesta

2- Bark Meow Animal sound

▼ Ejercicio 8 - Excepciones y Herencia

Considera el siguiente bloque de código:

```
import java.io.IOException;
import java.io.FileNotFoundException;
```

```
class Base {
    void show() throws IOException {
        System.out.println("Base show");
    }
}

class Derived extends Base {
    void show() throws FileNotFoundException {
        System.out.println("Derived show");
    }
}

public class Main {
    public static void main(String[] args) {
        Base obj = new Derived();
        try {
            obj.show();
        } catch (IOException e) {
            System.out.println("Exception caught");
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Base show
2- Derived show
3- Exception caught
4- Compilation error

Análisis de código

1. La clase `Base` tiene el método `show()` el cual lanza `IOException` .
2. La clase `Derived` hereda de la clase `Base` y hace Override al método `show()` lanzando una excepción subclase de `IOException` .
3. En el método `main()` se crea un objeto de tipo `Derived` y se le apunta desde una variable de tipo `Base` .
4. En un bloque try-catch se invoca al método `show()` . Hay que pensar en el tipo de objeto que invoca al método, por lo que el método `show()` es el correspondiente al que tiene la clase `Derived show` .

Respuestas

2- Derived show

▼ Ejercicio 9 - Concurrencia y Sincronización

Considera el siguiente bloque de código:

```
class SharedResource {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
```

```
    public synchronized void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}

class IncrementThread extends Thread {
    private SharedResource resource;

    public IncrementThread(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            resource.increment();
        }
    }
}

class DecrementThread extends Thread {
    private SharedResource resource;

    public DecrementThread(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            resource.decrement();
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();
        Thread t1 = new IncrementThread(resource);
        Thread t2 = new DecrementThread(resource);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(resource.getCount());
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- 1000
2- 0
3- -1000
4- Compilation error

Análisis del código

1. En la clase `SharedResource` se tiene el atributo `count = 0` , su respectivo `getCount()` y dos métodos que incluyen `synchronized` , `increment` incrementa `count` y `decrement` decrementa `count` .
2. Se tienen dos clases que heredan de `Thread` , `IncrementThread` y `DecrementThread` . Estas clases tienen como atributo un `SharedResource` , el cual se asigna mediante constructor, y se realiza el override del método `run()` indicando que se va a invocar el método `increment` o `decrement` dentro de un ciclo según el hilo que se ejecute.
3. El método `main()` instancia un `SharedResource` . Posteriormente crea un hilo para `IncrementThread` y `DecrementThread` pasando el mismo `SharedResource` a ambos.
4. Se inicializan los hilos.
5. Ambos ciclos for se compensan, de manera que cuando `IncrementThread` incrementa, `DecrementThread` decrementa en la misma proporción, por lo que el valor final de `count` es 0.
6. Finalmente con `join()` se reintegran los hilos y el programa finaliza imprimiendo 0.

Respuesta

2- 0

▼ Ejercicio 10 - Genericos y Excepciones

Considera el siguiente bloque de código:

```
class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() throws ClassCastException {
        if (item instanceof String) {
            return (T) item; // Unsafe cast
        }
        throw new ClassCastException("Item is not a String");
    }
}

public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setItem("Hello");

        try {
            String item = stringBox.getItem();
            System.out.println(item);
        } catch (ClassCastException e) {
```

```
            System.out.println("Exception caught");
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?

1- Hello
2- Exception caught
3- Compilation error
4- ClassCastException

Análisis de código

1. La clase Box es genérica y puede contener un objeto de cualquier tipo. Tiene como atributo una instancia genérica `item` . Cuenta con su método `setItem()` para establecer el valor del atributo `getItem()` y para devolver el atributo.
2. El método `getItem()` devuelve `item` únicamente si `item` es una instancia de `String` , de lo contrario se lanza `ClassCastException` .
3. Cast inseguro hace referencia a que no se tienen garantía de que `item` sea del tipo correcto y pueda lanzar una excepción `ClassCastException`.
4. El método `main()` instancia un `Box` con genérico `String` y se asigna el `item` .
5. Dentro de un bloque try-catch se extrae el `item` a través de `getItem()` y posteriormente se imprime.
6. No hay excepción por lo que se imprime `hello`

Respuesta

1- Hello

▼ Ejercicio 11 - Herencia y Cast

```
public class Main {
    public static void main(String[] args) {
            Padre objetoPadre = new Padre();
            Hija objetoHija = new Hija();
            Padre objetoHija2 = (Padre) new Hija();

            objetoPadre.llamarClase();
            objetoHija.llamarClase();
                        objetoHija2.llamarClase();

            Hija objetoHija3 = (Hija) new Padre();
            objetoHija3.llamarClase();
    }
}

public class Hija extends Padre {
    public Hija() {
        // Constructor de la clase Hija
    }

    @Override
```

```
    public void llamarClase() {
        System.out.println("Llame a la clase Hija");
    }
}

public class Padre {
    public Padre() {
        // Constructor de la clase Padre
    }

    public void llamarClase() {
        System.out.println("Llame a la clase Padre");
    }
}
```

¿Cuál es la respuesta correcta?

a) Llame a la clase Padre
Llame a la clase Hija
Llame a la clase Hija
Error: java.lang.ClassCastException

b) Llame a la clase Padre
Llame a la clase Hija
Llame a la clase Hija
Llame a la clase Hija

c) Llame a la clase Padre
Llame a la clase Hija
Llame a la clase Hija
Llame a la clase Padre

Análisis de código

1. La clase `Padre` tiene un constructor y un método `llamarClase()` .
2. La clase `Hija` hereda de `Padre` , tiene su constructor y hace Override de `llamarClase()` .
3. En la clase `main()` se crea un objeto de tipo `Padre` con una variable `Padre` . También se crea un objeto de tipo `Hija` en un contenedor tipo `Hija` . Finalmente se crea un objeto tipo `Hija` y haciendole un cast (innecesario) se almacena en una variables tipo `Padre` .
4. `objetoPadre` es tipo `Padre` en contenedor `Padre` y al llamar a `llamarClase()` se ejecuta `Llame a la clase Padre` .
5. `objetoHija` es tipo `Hija` en contenedor `Hija` y al llamar a `llamarClase()` se ejecuta `Llame a la clase hija` .
6. `objetoHija2` es tipo `Hija` en contenedor `Padre` y al llamar a `llamarClase()` se ejecuta `Llame a la clase hija` debido a que nos importa el tipo de objeto.
7. En la siguiente línea se crea un objeto `Padre` y se quiere almacenar en una variable `Hija` , es aqui donde sucede la excepción `ClassCastException` .

Respuesta

a) Llame a la clase Padre
Llame a la clase Hija
Llame a la clase Hija
Error: java.lang.ClassCastException

▼ Ejercicio 12 - Herencia y Cast

```
public class main {
    public static void main(String[] args) {
          Animal uno=new Animal();
          Animal dos=new Dog();
          uno.makeSound();
          dos.makeSound();
           Dog tres=(Dog)new Animal();
          tres.makeSound();
      }
}

class Animal {
      void makeSound() {
          System.out.println("Animal sound");
      }
}

class Dog extends Animal {
    void makeSound() {
          System.out.println("Wau Wau");
       }
}
```

1) Animal sound Wau Wau compilation error

2) Compilation Error

3) Animal sound Wau Wau Animal sound

4) Animal sound

Análisis de código

1. La clase `Animal` define el método `makeSound()` .
2. La clase `Dog` hereda de la clase `Animal` y hace Override del método `makeSound()` .
3. En el método `main()` se crea un objeto `uno` de tipo `Animal` en una variable `Animal` y un objeto `dos` de tipo `Dog` en una variable `Animal` .
4. Se invoca al método `makeSound()` de ambos objetos. El objeto `uno` imprime `Animal sound` y el objeto `dos` imprime `Wau Wau` .
5. Se intenta crear un objeto de tipo `Animal` en una variable de tipo `Dog` , por lo que se lanza `ClassCastException` .

Respuesta

Animal sound Wau Wau ClassCastException

▼ Ejercicio 13 - Sobrecarga y Tipos (Mutabilidad)

```
public class Ej13 {
    public static void main(String[] args) {
        Cambios uno = new Cambios();
        int x=1;
```

```
        String hola="hola";
        StringBuilder hola2=new StringBuilder("hola2");
        Integer x2=4;
        uno.makeSound(x, hola);
        uno.makeSound(x2, hola2);
        System.out.println("Cambios?: "+x+","+hola+","+x2+","+hola2);
    }
}

class Cambios{
   void makeSound(int x, String s) {
       s="cambiando string";
       x=5;
   }
   void makeSound(Integer x,StringBuilder s) {
        x=9;
        s=s.delete(0,s.length());
   }
}
```

1) Compilation error
2) Cambios?: 1,hola,4,
3) Cambios?: 1,hola,4,hola2
4) Cambios?: 5,cambiando string,9

Análisis de código

1. La clase `Cambios` hace Overload en el método `makeSound()` .
2. En el método `main()` se crea un objeto de tipo `Cambios` , la variable `x=1` , `hola="hola"` , la variable `hola2="hola2"` de tipo `StringBuilder` , la variable `x2=4` de tipo `Integer` .
3. Se invoca al método `makeSound()` desde el objeto `uno` .
4. La primera invocación corresponde al método `makeSound(int x, String s)` pues los parámetros de ajustan bien.
5. En el método `x` hace referencia a una variable local del método, por lo que el valor de `x` no se modifica.
6. De igual manera `s` es inmutable y es una referencia local del método, por lo que su valor no modifica a la del método `main()` .
7. La segunda invocación corresponde al método `makeSound(Integer x, StringBuilder s)` .
8. `Integer` es inmutable y `x` hace referencia a una variable local de método, por lo que su valor no modifica a la del método `main()` .
9. `StringBuilder` es mutable, por lo que la operación sobre su referencia sí lo modifica.
10. Finalmente se imprime `Cambios?: 1, hola,4,`

▼ Ejercicio 14 - Excepciones y Herencia

```
class Animal {
   void makeSound() throws Exception {
      System.out.println("Animal makes a sound");
   }
}
class Dog extends Animal {
```

```
    void makeSound() throws RuntimeException {
       System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
       Animal myDog = new Dog();
       try {
           myDog.makeSound();
        } catch (Exception e) {
           System.out.println("Exception caught");
        }
    }
}
```

¿Cuál sería la salida en consola al ejecutar este código?
1- Dog barks
2- Animal makes a sound
3- Exception caught
4- Compilation error

Análisis de código

1. La clase `Animal` define el método `makeSound()` que lanza una `Exception` .
2. La clase `Dog` hereda de `Animal` y hace Override al método `makeSound()` y lanza `RuntimeException` . Recordar que no es necesario manejar unchecked.
3. En el método `main()` se crea un objeto de tipo `Dog` en una variable `Animal` .
4. En un bloque try-catch se invoca al método `makeSound()` a través de `myDog` .
5. Nunca se presenta una excepción y se imprime `Dog barks` .

Respuesta

1- Dog barks

▼ Ejercicio 15 - Split y Expresiones regulares

```
class main {
    public static void main(String[] args) {
        String str = "1a2b3c4d5e6f";
        String[] splitStr = str.split("\\D");

        for(String elemento : splitStr){
            System.out.println(elemento);
        }
    }
}
```

Análisis de código

1. En el método `main()` se crea una cadena con `1a2b3c4d5e6f` .

2. Posteriormente se utiliza el método `split()` . Este método pertenece a la clase `String` y toma como argumento una expresión regular (regex).
3. `split()` divide la cadena en un arreglo de cadenas basado en la expresión regular proporcionada.
4. La expresión regular `\\D` representa cualquier carácter que no sea un dígito (opuesto a `\\d`, que representa cualquier dígito). Además el doble backslash se utiliza para evitar el escape por uno simple.
5. Entonces `split("\\D")` dividira la cadena en partes donde encuentre caracteres que no sean dígitos. En este caso los caracteres como `a` `b` `c` , etc. serán los delimitadores.
6. Entonces el arregla contiene `[1,2,3,4,5,6]` ya que los caracteres delimitadores se omiten.