Conceptos Clave de Teoría

Encapsulación

Wrappers-> Unboxing y Autoboxing

Referencias

Strings y String Pool

Método equals

Reservar Memoria Para Arrays

Que TD es iterable y cual no

Herencia

Clase Abstracta

Polimorfismo-> downcasting upcasting

Herencia Multiple en Interfaces

Excepciones extienden de Exception

HashMaps

Recorrerlo usando las claves:

```
public class Main{
    private HashMap<Integer, String> mapa;

    public void main(String[] args){
        mapa= new HashMap<>();

        Set<Integer> claves = mapa.keySet();
        for(Integer int: claves){
            String str = mapa.get(int);
            System.out.println(str);
        }
    }
}
```

Recorrerlo usando los valores

```
public class Main{
    private HashMap<Integer, String> mapa;

    public void main(String[] args){
        mapa= new HashMap<>();
```

```
        Collection<String> valores = mapa.values();
        for(String str: valores){
            System.out.println(str);
        }
    }
}
```

Recorrerlo usando la dupla de valores (mapa de entrada)

```
public class Main{
    private HashMap<Integer, String> mapa;

    public void main(String[] args){
        mapa= new HashMap<>();

        Set<Map.Entry<Integer,String>> valores = mapa.entrySet();
        for(Map.Entry dupla: valores){
            String str = dupla.getValue(); //dupla.getKey();
            System.out.println(str);
        }
    }
}
```

Recorrerlo Usando Iterator

```
public class Main{
    private HashMap<Integer, String> mapa;

    public void main(String[] args){
        mapa= new HashMap<>();

        Set<Integer> valores = mapa.keySet();
        Iterator<Integer> it = valores.iterator();

        while(it.hasNext()){
            String str = it.next();
            System.out.prinln(str);
        }
    }
}
```

Jerarquías

Aquí tienes el código completo que refleja los principios de jerarquía de clases, polimorfismo, reutilización de métodos de la clase padre y correcta implementación de constructores:

```java
public abstract class Vehiculo {
    private String nombre;
    private int valor;

    // Constructor sin parámetros
    public Vehiculo() {
    }

    // Constructor con parámetros
    public Vehiculo(String nombre, int valor) {
        this.nombre = nombre;
        this.valor = valor;
    }

    // Métodos getter y setter
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getValor() {
        return valor;
    }

    public void setValor(int valor) {
        this.valor = valor;
    }

    // Método abstracto que será implementado por las clases hijas
    public abstract int calcularValor();

    // Método común para imprimir detalles del vehículo
    public void imprimirDetalles() {
        System.out.println("Nombre: " + nombre + ", Valor base: " + valor);
    }
}
```

```java
public class Coche extends Vehiculo {
    private boolean desembrague;

    // Constructor sin parámetros
    public Coche() {
        super();
    }
```

```java
    // Constructor con parámetros
    public Coche(String nombre, int valor, boolean desembrague) {
        super(nombre, valor);
        this.desembrague = desembrague;
    }

    // Getter y setter
    public boolean isDesembrague() {
        return desembrague;
    }

    public void setDesembrague(boolean desembrague) {
        this.desembrague = desembrague;
    }

    // Implementación del método abstracto
    @Override
    public int calcularValor() {
        return getValor() + (desembrague ? 1000 : 0); // Valor adicional si tiene desembrague
    }

    // Sobrescritura del método imprimirDetalles
    @Override
    public void imprimirDetalles() {
        super.imprimirDetalles();
        System.out.println("Desembrague: " + (desembrague ? "Sí" : "No"));
    }
}
```

```java
public class Moto extends Vehiculo {
    private boolean anchoDeManubrio;

    // Constructor sin parámetros
    public Moto() {
        super();
    }

    // Constructor con parámetros
    public Moto(String nombre, int valor, boolean anchoDeManubrio) {
        super(nombre, valor);
        this.anchoDeManubrio = anchoDeManubrio;
    }

    // Getter y setter
    public boolean isAnchoDeManubrio() {
        return anchoDeManubrio;
    }
```

```
    public void setAnchoDeManubrio(boolean anchoDeManubrio) {
        this.anchoDeManubrio = anchoDeManubrio;
    }

    // Implementación del método abstracto
    @Override
    public int calcularValor() {
        return getValor() + (anchoDeManubrio ? 500 : 0); // Valor adicional si tiene manubrio
    }

    // Sobrescritura del método imprimirDetalles
    @Override
    public void imprimirDetalles() {
        super.imprimirDetalles();
        System.out.println("Manubrio ancho: " + (anchoDeManubrio ? "Sí" : "No"));
    }

    public String hacerElsubnormal(){
        System.out.println("PuTA PSOE");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Vehiculo V1 = new Coche("Toyota", 20000, true);//Upcasting
        Vehiculo V2 = new Moto("Yamaha", 15000, false);

        V1.imprimirDetalles();
        System.out.println("Valor total del coche: " + coche.calcularValor());

        V2.hacerElsubnormal(); //dará warning y no compila

        Moto moto = (Moto) V2; //downcasting
        moto.hacerElsubnormal(); //lo ejecutará sin problema
    }
}
```

Excepciones

```
// Clase de excepción personalizada
public class VehiculoException extends Exception {
    public VehiculoException(String mensaje) {
        super(mensaje);
    }
    // Método adicional para ayudar a resolver el problema
```

```
    public String sugerirSolucion() {
        return "Revise los parámetros de entrada y asegúrese de que sean válidos.";
    }
}
```

En vehiculo:

```
public Vehiculo(String nombre, int valor) throws VehiculoException {
    if (valor < 0) {
        throw new VehiculoException("El valor del vehículo no puede ser negativo.");
    }
    this.nombre = nombre;
    this.valor = valor;
}

public void setValor(int valor) throws VehiculoException {
    if (valor < 0) {
        throw new VehiculoException("El valor del vehículo no puede ser negativo.");
     }
     this.valor = valor;
}
```

```
// Clase principal para demostrar manejo de excepciones
public class Main { public static void main(String[] args) {
    try {
        // Intentamos crear un vehículo con un valor negativo
        Vehiculo coche = new Coche("Toyota", -20000, true);
    } catch (VehiculoException e) {
        // Capturamos la excepción y mostramos un mensaje
        System.out.println("Error al crear el vehículo: " + e.getMessage());
        System.out.println("Sugerencia: " + e.sugerirSolucion());
    } try {
    // Crear un vehículo correctamente
        Vehiculo moto = new Coche("Honda", 15000, false); // Intentar asignar un valor negativo
        moto.setValor(-5000);
    } catch (VehiculoException e) { // Capturar y manejar la excepción
        System.out.println("Error al actualizar el valor del vehículo: " + e.getMessage());
        System.out.println("Sugerencia: " + e.sugerirSolucion()); // Opcionalmente, podemos vol
        throw new RuntimeException("Excepción no manejada", e);
    }
        System.out.println("Programa finalizado.");
    }
}
```

Interfaces

```java
// Definición de la interfaz
public interface VehiculoInterface {
    // Método abstracto
    void mover();

    // Método por defecto
    default void encender() {
        System.out.println("El vehículo está encendido.");
    }

    // Método estático
    static void mostrarTipoVehiculo() {
        System.out.println("Todos los vehículos tienen ruedas.");
    }
}

// Clase Coche que implementa la interfaz
public class Coche implements VehiculoInterface {
    private String modelo;

    public Coche(String modelo) {
        this.modelo = modelo;
    }

    @Override
    public void mover() {
        System.out.println("El coche " + modelo + " se está moviendo.");
    }
}

// Clase Moto que implementa la interfaz
public class Moto implements VehiculoInterface {
    private String marca;

    public Moto(String marca) {
        this.marca = marca;
    }

    @Override
    public void mover() {
        System.out.println("La moto " + marca + " se está moviendo.");
    }

    // Sobrescribiendo el método por defecto
    @Override
    public void encender() {
        System.out.println("La moto está encendida de manera personalizada.");
```

```
    }
}

// Clase principal para demostrar el uso de la interfaz
public class Main {
    public static void main(String[] args) {
        // Uso del polimorfismo: referencias de tipo VehiculoInterface
        VehiculoInterface coche = new Coche("Toyota");
        VehiculoInterface moto = new Moto("Yamaha");

        // Métodos abstractos
        coche.mover();
        moto.mover();

        // Métodos por defecto
        coche.encender();
        moto.encender();

        // Método estático de la interfaz
        VehiculoInterface.mostrarTipoVehiculo();

        // Uso adicional para demostrar independencia
        realizarAccion(coche);
        realizarAccion(moto);
    }

    // Método que usa polimorfismo para trabajar con VehiculoInterface
    public static void realizarAccion(VehiculoInterface vehiculo) {
        System.out.println("Acción genérica para cualquier vehículo:");
        vehiculo.mover();
    }
}
```