Resumo para o Exame de Java

✅ Fundamentos de Java

- Tipos de datos básicos en Java:
`int`, `double`, `boolean`, `char`, `String`.

- Estruturas de control: `if`, `else`, `switch`, `while`, `for`.

✅ Modificadores de Acceso

```
class Exemplo {
    public int publico;
    protected int protexido;
    int porDefecto;
    private int privado;
}
```

✅ Clases, Interfaces e Herdanza

Clase abstracta vs Interface:

- 🔷 **Clase abstracta** → Usa cando queres definir un comportamento base que poida ser herdado polas subclases.
- 🔷 **Interface** → Usa cando queres definir un contrato que diferentes clases poden implementar, incluso se non están relacionadas.

✅ Usa unha **clase abstracta** cando:

- Queres compartir código entre subclases.
- Queres que as subclases herden comportamento por defecto.

✅ Usa unha **interface** cando:

- Queres definir un comportamento que clases non relacionadas poden implementar.
- Queres permitir múltiples implementacións (Java permite implementar varias interfaces).

```
interface Alugable {
    boolean alugar();
    boolean devolver();
    double prezo();
}
```

```
abstract class Vehiculo implements Alugable, Comparable {
    protected String matricula;
    protected String marca;
    protected String modelo;

    @Override
    public int compareTo(Vehiculo outro) {
        return this.matricula.compareTo(outro.matricula);
    }

    @Override
    public abstract boolean alugar();
    @Override
    public abstract boolean devolver();
    @Override
    public abstract double prezo();
}
```

✅ Xenéricos

```
class Caixa {
    private T valor;

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

    public T getValor() {
        return valor;
    }
}

Caixa caixa = new Caixa<>();
caixa.setValor(42);
System.out.println(caixa.getValor()); // 42
```

✅ Collection Framework COMPLETO

Java proporciona o framework `Collection` para xestionar grupos de datos. Aquí tes unha comparativa das interfaces máis comúns:

Interface	Permite duplicados?	Mantén orde?	Implementacións comúns	Comentario

`List`	✔	✔	ArrayList, LinkedList	Acceso rápido por índice
`Set`	✘	✘ (excepto TreeSet)	HashSet, TreeSet	Sen duplicados
`Map`	✔ (en valores) ✘ (en claves)	✘ (excepto TreeMap)	HashMap, TreeMap	Almacena pares clave-valor

Máis sobre as coleccións:

- `LinkedList` é útil cando precisas moitas insercións e eliminacións no medio da lista.
- `TreeSet` ou `TreeMap` son opcións axeitadas cando necesitas elementos automaticamente ordenados.
- `HashSet` ou `HashMap` son excelentes cando o rendemento é crucial e a orde non importa.
- `PriorityQueue` é útil para tarefas que precisan traballar con elementos en orde de prioridade.

Táboas rápidas:

Métodos comúns de List

Método	**Descrición**
`add(E e)`	Engade un elemento
`get(int index)`	Devolve o elemento nun índice
`set(int index, E element)`	Substitúe o elemento nun índice
`remove(int index)`	Elimina o elemento nun índice
`size()`	Devolve o número de elementos

Métodos comúns de Set

Método	**Descrición**
`add(E e)`	Engade un elemento (false se xa existe)

`remove(Object o)`	Elimina un elemento
`contains(Object o)`	Devolve true se o elemento está presente
`size()`	Devolve o número de elementos

Métodos comúns de Map

Método	**Descrición**
`put(K key, V value)`	Engade ou substitúe un par clave-valor
`get(Object key)`	Devolve o valor asociado á clave
`remove(Object key)`	Elimina unha entrada pola súa clave
`containsKey(Object key)`	Comproba se a clave existe
`size()`	Devolve o número de entradas

✅ Streams e Lambdas

```
List numeros = List.of(1, 2, 3, 4, 5);
numeros.stream()
       .filter(n -> n % 2 == 0)
       .map(n -> n * 2)
       .forEach(System.out::println);
```

✅ Clase Almacen

```
class Almacen {
    private List elementos = new ArrayList<>();

    public T add(T data) throws AlmacenException {
        if (elementos.contains(data)) {
            throw new AlmacenException("Elemento xa existe");
        }
        elementos.add(data);
        return data;
    }
}
```

✅ Validator

```
class Validator {
    public static boolean validarDNI(String dni) {
        return dni.matches("\d{8}[A-HJ-NP-TV-Z]");
    }

    public static int validarInputInt(String input) throws NumberFormatException {
        return Integer.parseInt(input);
    }

    public static double validarInputDouble(String input) throws NumberFormatException {
        return Double.parseDouble(input);
    }

    public static boolean validarEmail(String email) {
        return email.matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$\");
    }
}
```

✅ Comparator

✅ Usa `Comparable` para definir unha orde natural:

```
class Persona implements Comparable {
    String nome;

    @Override
    public int compareTo(Persona outra) {
        return this.nome.compareTo(outra.nome);
    }
}
```

✅ Usa `Comparator` para definir orde personalizada:

```
Comparator comparadorPorLongitud = (p1, p2) -> p1.nome.length() - p2.nome.length();
```

✅ Excepcións

```
class AlmacenException extends Exception {
    public AlmacenException(String mensaje) {
        super(mensaje);
    }
}
```

✅ Clases Internas e Anónimas

```
Runnable tarefa = () -> System.out.println("Clase anónima executada");
tarefa.run();
```

✅ Iterable vs Iterator

Iterable: Define que unha clase pode ser percorrida usando un bucle `for-each` .

```
List lista = List.of("A", "B", "C");
for (String elemento : lista) {
    System.out.println(elemento);
}
```

Iterator: Permite percorrer manualmente unha colección.

```
Iterator iterador = lista.iterator();
while (iterador.hasNext()) {
    System.out.println(iterador.next());
}
```

✅ Algoritmos Comúns

Ordenación:

```
Collections.sort(lista);
```

Busca:

```
int index = Collections.binarySearch(lista, "Carlos");
```

Máximo/Mínimo:

```
String max = Collections.max(lista);
String min = Collections.min(lista);
```

✅ Streams Avanzados

reduce:

```
int suma = lista.stream().reduce(0, Integer::sum);
```

collect:

```
List listaMaiusculas = lista.stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

flatMap:

```
List> listOfLists = List.of(
    List.of("A", "B"),
    List.of("C", "D")
);

listOfLists.stream()
    .flatMap(List::stream)
    .forEach(System.out::println);
```

✅ Menú no Main

```
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int opcion;

        do {
            System.out.println("Menú de Aluguer:");
            System.out.println("1. Engadir novo vehículo");
            System.out.println("2. Buscar vehículo por matrícula");
            System.out.println("3. Alugar un vehículo");
            System.out.println("4. Listar vehículos dispoñibles");
            System.out.println("5. Listar alugueres en curso");
            System.out.println("6. Rexistrar unha entrega");
            System.out.println("0. Saír");
            System.out.print("Elixe unha opción: ");

            opcion = scanner.nextInt();
            scanner.nextLine(); // Consumir o salto de liña

            switch (opcion) {
    case 1:
        engadirNovoVehiculo();
        break;
    case 2:
        buscarVehiculoPorMatricula();
        break;
    case 3:
        alugarVehiculo();
```

```
            break;
        case 4:
            listarVehiculosDispoñibles();
            break;
        case 5:
            listarAlugueresEnCurso();
            break;
        case 6:
            rexistrarEntrega();
            break;
        case 0:
            System.out.println("Saíndo do menú...");
            break;
        default:
            System.out.println("Opción non válida. Tenta de novo.");
    }
        } while (opcion != 0);

        scanner.close();
    }
}
```

- Analiza o enunciado do exame e identifica que partes do exercicio valen máis puntos. Comeza sempre por esas para garantir unha maior puntuación.

- Organiza a túa solución antes de escribir código. Pensa na estrutura das clases, os métodos necesarios e os datos que vas almacenar. Podes facer un pequeno esquema en papel ou en texto antes de poñerte a programar.

- Comeza implementando as clases máis importantes e as interfaces que van definir o comportamento básico. Por exemplo, establece primeiro a interface "Alugable" e as clases base como "Vehiculo".

- Crea un esqueleto inicial do programa. Podes incluír un menú simple que te permita engadir vehículos, buscar por matrícula, alugar, listar dispoñibles e rexistrar entregas. Isto axudarache a comprobar se os métodos básicos funcionan correctamente.

- Traballa en pequenos incrementos. Proba unha funcionalidade, verifica que funcione e despois avanza á seguinte. Isto axudará a evitar erros que se propaguen.

- Emprega exemplos sinxelos para validar cada parte. Por exemplo, crea uns poucos vehículos de proba e verifica que podes buscalos, alugalos e devolvelos.

- Antes de entregar, revisa o código en busca de posibles erros, probas que falten ou funcionalidades que quedaron a medias. Asegúrate de que o menú cubra todas as operacións necesarias e que todo compile sen erros.