FEIP

Creada por	Ⓦ Wilmar
Hora de creación	@4 de junio de 2025 15:33
Categoría	Notas Resumen de Módulos
Fecha de última actualización	@4 de julio de 2025 11:25
Última actualización realizada por	Ⓦ Wilmar

Excepciones básicas en Java que conviene memorizar por frecuencia y relevancia:

Checked Exceptions (obligan a usar `try-catch` o `throws`)

1. **IOException** – Fallos de entrada/salida (archivos, streams).
2. **FileNotFoundException** – Archivo no encontrado (subtipo de IOException).
3. **SQLException** – Errores en operaciones con bases de datos.
4. **ParseException** – Error al interpretar texto (por ejemplo, fechas).
5. **ClassNotFoundException** – Clase no encontrada al cargar dinámicamente.

Unchecked Exceptions (subclases de RuntimeException, no requieren manejo obligatorio)

1. **ArithmeticException** – División por cero, operaciones aritméticas inválidas.
2. **NullPointerException** – Acceso a un objeto nulo.
3. **ArrayIndexOutOfBoundsException** – Acceso a índice inválido en un array.
4. **NumberFormatException** – Conversión inválida de String a número.
5. **IllegalArgumentException** – Argumento ilegal pasado a un método.
6. **IllegalStateException** – Estado ilegal del objeto para una operación.
7. **ClassCastException** – Conversión inválida de tipos.

Aprendé especialmente las diferencias entre `checked` y `unchecked`, y ubicá ejemplos típicos para cada una.

Diferencias entre checked y unchecked exceptions en Java:

1. Checked Exceptions

- **Definición**: Son excepciones que el compilador *obliga* a manejar (usando `try-catch` o `throws`).
- **Jerarquía**: Subclases de `Exception`, pero **no** de `RuntimeException`.
- **Manejo**: Obligatorio. Si no se manejan, el programa **no compila**.
- **Ejemplos comunes**:
  - `IOException`
  - `SQLException`
  - `ParseException`
  - `ClassNotFoundException`

```
public void leerArchivo() throws IOException {
    FileReader fr = new FileReader("archivo.txt"); // debe manejarse
}
```

2. Unchecked Exceptions

- **Definición**: Son excepciones que el compilador *no obliga* a manejar.
- **Jerarquía**: Subclases de `RuntimeException`.
- **Manejo**: Opcional. Si no se manejan, el programa compila pero puede fallar en tiempo de ejecución.
- **Ejemplos comunes**:
  - `NullPointerException`
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
  - `NumberFormatException`

```
public void dividir(int a, int b) {
    int resultado = a / b; // puede lanzar ArithmeticException
}
```

Resumen clave

Característica	Checked	Unchecked
Jerarquía	Exception (no RuntimeException)	RuntimeException
Verificación en compilación	Sí	No
Necesidad de manejo	Obligatorio	Opcional
Causa típica	Fallos externos	Errores de programación

Las `checked exceptions` fuerzan a prever errores externos controlables. Las `unchecked` reflejan errores lógicos internos que deberían evitarse con código correcto.

💡 Estructuras de Datos y Algoritmos

- Métodos **CRUD clásicos** (como `.add()`, `.remove()`, `.get()`, etc.).
- Métodos **complementarios relevantes** (como `.contains()`, `.sort()`, `.keySet()`, etc.).
- Ejemplos de **algoritmos manuales**: ordenamientos (burbuja, selección) y búsquedas (lineal, binaria).
- En cada caso: **descripción corta + ejemplo de uso.**

🔧 1. Object (java.lang.Object)

Un **objeto** en Java es una instancia de una clase que encapsula datos (atributos) y comportamientos (métodos). Es la unidad básica de programación orientada a objetos.

Ejemplo:

```
Persona p = new Persona("Juan", 30);
```

Método	Descripción	Ejemplo
`.equals(obj)`	Compara si dos objetos son lógicamente iguales	`a.equals(b)`
`.toString()`	Representación String del objeto	`System.out.println(obj.toString())`
`.hashCode()`	Devuelve código hash del objeto	`int hash = obj.hashCode();`

📌 2. String (`java.lang.String`)

String representa una secuencia inmutable de caracteres. Es ampliamente usado para manejar texto. Debido a su inmutabilidad, cada operación que lo modifica crea un nuevo objeto.

Ejemplo:

```
String saludo = "Hola";
```

Método	Descripción	Ejemplo
`.charAt(i)`	Devuelve el carácter en la posición `i`	`"hola".charAt(1)` → `'o'`
`.length()`	Largo de la cadena	`"hola".length()` → `4`
`.substring(i,j)`	Substring entre `i` (incluido) y `j` (excluido)	`"hola".substring(1,3)` → `"ol"`
`.equals()`	Compara strings por valor	`"a".equals("a")` → `true`
`.contains()`	Verifica si contiene una secuencia de caracteres	`"hola".contains("la")` → `true`
`.indexOf()`	Posición del primer match	`"abcabc".indexOf("b")` → `1`
`.toLowerCase()`	Convierte a minúsculas	`"HOLA".toLowerCase()` → `"hola"`
`.split(" ")`	Divide string por separador	`"a,b".split(",")` → `["a", "b"]`

📚 3. Arrays (`int[]` , `String[]` , etc.) + `Arrays` class

Un **array** es una estructura fija que almacena elementos del mismo tipo en posiciones contiguas de memoria. Su tamaño se define al momento de su creación.

Ejemplo:

```
int[] numeros = {1, 2, 3};
```

Método o Técnica	Descripción	Ejemplo
`arr[i]`	Acceso directo a valor	`int x = arr[0];`
`arr.length`	Largo del array	`for(int i = 0; i < arr.length; i++) {...}`
`Arrays.sort(arr)`	Ordena el array (QuickSort interno)	`Arrays.sort(arr);`
`Arrays.binarySearch()`	Búsqueda binaria (requiere array ordenado)	`Arrays.binarySearch(arr, 4)`
Manual Bubble Sort	Algoritmo de ordenamiento por burbuja	Ver más abajo
Manual Linear Search	Recorrer y comparar uno por uno	Ver más abajo

📋 4. List (Interface) – `ArrayList` , `LinkedList`

List es una colección ordenada que permite elementos duplicados. Es dinámica (a diferencia del array) y permite acceso por índice.

Implementaciones comunes: ArrayList, LinkedList.

Ejemplo:

```
List<String> lista = new ArrayList<>();
```

Método	Descripción	Ejemplo
`.add(e)`	Agrega un elemento	`lista.add("hola");`
`.add(i, e)`	Inserta en posición específica	`lista.add(1, "mundo");`
`.get(i)`	Devuelve elemento en índice `i`	`lista.get(0);`
`.set(i, e)`	Reemplaza el valor en índice `i`	`lista.set(1, "nuevo");`
`.remove(i)`	Elimina por índice	`lista.remove(0);`

Método	Descripción	Ejemplo
`.contains(e)`	Verifica existencia	`lista.contains("hola");`
`.size()`	Cantidad de elementos	`int n = lista.size();`
`.clear()`	Elimina todos los elementos	`lista.clear();`
`Collections.sort(lista)`	Ordena lista	`Collections.sort(lista);`

ArrayList (`java.util.ArrayList`)

Una implementación de `List` basada en arrays dinámicos. Ofrece acceso rápido por índice pero inserciones/eliminaciones intermedias pueden ser costosas.

Ejemplo: `ArrayList<Integer> lista = new ArrayList<>();`

LinkedList (`java.util.LinkedList`)

Otra implementación de `List`, basada en nodos enlazados. Tiene buen rendimiento para inserciones/eliminaciones frecuentes, pero acceso más lento por índice.

Ejemplo: `LinkedList<String> cola = new LinkedList<>();`

🔢 5. Set (Interface) – `HashSet` , `TreeSet`

Set representa una colección sin elementos duplicados. No garantiza orden (en HashSet) pero sí puede hacerlo (LinkedHashSet, TreeSet).

Ejemplo:

```
Set<String> conjunto = new HashSet<>();
```

Método	Descripción	Ejemplo
`.add(e)`	Agrega (sin duplicados)	`set.add("hola");`
`.remove(e)`	Elimina si existe	`set.remove("hola");`
`.contains(e)`	Verifica existencia	`set.contains("hola");`
`.size()`	Tamaño del conjunto	`set.size();`
`.clear()`	Limpia el conjunto	`set.clear();`
`.isEmpty()`	Verifica si está vacío	`set.isEmpty();`

HashSet (`java.util.HashSet`)

Una implementación de `Set` que no garantiza orden de los elementos y ofrece acceso rápido mediante hash. No permite elementos duplicados.

Ejemplo: `HashSet<String> valores = new HashSet<>();`

📦 6. Map (Interface) – `HashMap` , `TreeMap`

`Map` representa una estructura de clave-valor. Cada clave es única, y se accede a los valores a través de sus claves.

Ejemplo: `Map<String, Integer> edades = new HashMap<>();`

Método	Descripción	Ejemplo
`.put(k, v)`	Inserta o actualiza valor asociado a clave	`map.put("clave", "valor");`
`.get(k)`	Devuelve el valor asociado a clave	`map.get("clave");`
`.remove(k)`	Elimina entrada por clave	`map.remove("clave");`
`.containsKey(k)`	Verifica si la clave existe	`map.containsKey("clave");`
`.containsValue(v)`	Verifica si el valor existe	`map.containsValue("valor");`
`.keySet()`	Devuelve conjunto de claves	`for(String k : map.keySet())`
`.values()`	Devuelve colección de valores	`for(String v : map.values())`
`.entrySet()`	Devuelve pares clave-valor	`for(Map.Entry<K,V> e : map.entrySet()) {...}`

HashMap (`java.util.HashMap`)

Una implementación de Map que usa una tabla hash. No garantiza orden, pero ofrece acceso rápido a los elementos mediante sus claves.

Ejemplo:

```
HashMap<String, String> mapa = new HashMap<>();
```

🛻 7. Vector (`java.util.Vector`)

Estructura similar a ArrayList pero **sincrónica** (thread-safe). Hoy se prefiere ArrayList en entornos no concurrentes por rendimiento.

Ejemplo:

```
Vector<Integer> vec = new Vector<>();
```

Método	Descripción	Ejemplo
`.add(e)`	Agrega elemento al final	`vector.add(10);`
`.get(i)`	Devuelve el elemento en posición `i`	`vector.get(0);`
`.remove(i)`	Elimina por índice	`vector.remove(1);`
`.size()`	Devuelve la cantidad de elementos	`vector.size();`
`.clear()`	Elimina todos los elementos	`vector.clear();`

🔁 8. Collection (Interfaz común a List, Set, etc.)

Método	Descripción	Ejemplo
`.add(e)`	Agrega un elemento	`col.add("dato");`
`.remove(e)`	Elimina un elemento	`col.remove("dato");`
`.contains(e)`	Verifica existencia	`col.contains("dato");`
`.isEmpty()`	Verifica si está vacía	`col.isEmpty();`
`.clear()`	Vacía la colección	`col.clear();`
`.iterator()`	Devuelve un iterador	`Iterator it = col.iterator();`

🥞 9. Stack (`java.util.Stack`)

Stack es una estructura **LIFO** (último en entrar, primero en salir). Se usa para problemas como expresiones, backtracking, navegación, etc.

Ejemplo:

```
Stack<Integer> pila = new Stack<>();
```

Método	Descripción	Ejemplo
`.push(e)`	Inserta un elemento en la cima	`stack.push(10);`
`.pop()`	Elimina y retorna el elemento de la cima	`int x = stack.pop();`

Método	Descripción	Ejemplo
`.peek()`	Devuelve el elemento de la cima sin eliminarlo	`int x = stack.peek();`
`.isEmpty()`	Verifica si la pila está vacía	`stack.isEmpty();`
`.search(e)`	Retorna la posición desde la cima (1 = top)	`stack.search(10);`

🔁 10. Queue (Interfaz) – `LinkedList`, `PriorityQueue` (`java.util.Queue`)

Queue es una estructura **FIFO** (primero en entrar, primero en salir), útil para manejar procesos en orden, tareas en cola, etc.

Ejemplo:

```
Queue<String> cola = new LinkedList<>();
```

Método	Descripción	Ejemplo
`.offer(e)`	Inserta un elemento al final (cola)	`queue.offer("A");`
`.poll()`	Elimina y retorna el primer elemento	`String x = queue.poll();`
`.peek()`	Devuelve el primer elemento sin eliminarlo	`String x = queue.peek();`
`.isEmpty()`	Verifica si la cola está vacía	`queue.isEmpty();`
`.size()`	Devuelve el tamaño de la cola	`queue.size();`

🔷 *Nota:* Aunque `.add()`, `.remove()` y `.element()` también existen, `.offer()`, `.poll()`, y `.peek()` son preferidos porque no lanzan excepciones al fallar.

🔃 11. Deque (Double Ended Queue – `ArrayDeque`, `LinkedList`) (`java.util.Queue`)

Deque (double-ended queue) permite inserciones y eliminaciones por ambos extremos, lo que la hace versátil como pila o cola.

Ejemplo:

```
Deque<Integer> deque = new ArrayDeque<>();
```

Método	Descripción	Ejemplo
`.addFirst(e)`	Agrega al inicio	`deque.addFirst(1);`
`.addLast(e)`	Agrega al final	`deque.addLast(2);`
`.removeFirst()`	Quita el primero	`int x = deque.removeFirst();`
`.removeLast()`	Quita el último	`int x = deque.removeLast();`
`.peekFirst()`	Mira el primero sin quitar	`deque.peekFirst();`
`.peekLast()`	Mira el último sin quitar	`deque.peekLast();`
`.isEmpty()`	Verifica si está vacía	`deque.isEmpty();`

🔷 *Deque permite simular tanto una pila como una cola.*

🔄 12. Iterator (de `Collection`) (`java.util.Iterator`)

El `Iterator` permite recorrer colecciones de forma segura, especialmente cuando se eliminan elementos durante la iteración.

Ejemplo:

```
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
    String val = it.next();
}
```

Métodos del `Iterator`

Método	Descripción	Ejemplo
`.hasNext()`	Verifica si hay más elementos	`while(it.hasNext())`
`.next()`	Devuelve el siguiente elemento	`String x = it.next();`
`.remove()`	Elimina el último elemento retornado por `.next()`	`it.remove();`

Ejemplo básico

```
List<String> lista = new ArrayList<>();
lista.add("a");
lista.add("b");
lista.add("c");
```

```
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
    String valor = it.next();
    if(valor.equals("b")) {
        it.remove(); // Seguro: elimina "b"
    }
}
```

🔷 *Nunca uses* `.remove()` *directamente sobre la colección dentro del* `for-each`*, usá* `Iterator` *para evitar* `ConcurrentModificationException`.

🧠 13. Algoritmos manuales

🔍 Búsqueda Lineal (Linear Search)

Recorre la lista comparando cada elemento con el buscado.

Uso: Simple y universal, pero lento.

```
int buscar(int[] arr, int x) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] == x) return i;
    }
    return -1;
}
```

🔍 Búsqueda Binaria (Binary Search - requiere array ordenado)

Requiere un array ordenado. Divide la búsqueda a la mitad en cada paso.

Uso: Muy eficiente (O(log n))

```
int binaria(int[] arr, int x) {
    int i = 0, j = arr.length - 1;
    while(i <= j) {
        int m = (i + j) / 2;
        if(arr[m] == x) return m;
        else if(arr[m] < x) i = m + 1;
        else j = m - 1;
    }
```

```
    return -1;
}
```

🔃 Ordenamiento Burbuja (Bubble Sort)

Compara pares adyacentes y los intercambia si están en orden incorrecto. Se repite hasta que no hay más cambios.

Uso: Simple pero ineficiente en grandes volúmenes.

```
void burbuja(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        for(int j = 0; j < arr.length - 1 - i; j++) {
            if(arr[j] > arr[j+1]) {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

🔃 Ordenamiento por Selección (Selection Sort)

Busca el valor mínimo y lo coloca en su posición final, repitiendo para cada elemento.

Uso: Más eficiente que burbuja, pero aún lento para listas grandes.

```
void seleccion(int[] arr) {
    for(int i = 0; i < arr.length; i++) {
        int min = i;
        for(int j = i + 1; j < arr.length; j++) {
            if(arr[j] < arr[min]) min = j;
        }
        int tmp = arr[i];
        arr[i] = arr[min];
        arr[min] = tmp;
```

```
    }
}
```

⚙️ Insertion Sort (Ordenamiento por Inserción)

Toma elementos uno por uno y los "inserta" en la posición correcta del lado izquierdo ya ordenado.

Uso: Muy bueno para listas pequeñas o casi ordenadas.

```
for (int i = 1; i < arr.length; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}

int[] arr = {3, 4, 5, 1};
```

✅ ¿Qué debes importar en cada caso?

Estructura	Importación necesaria
`ArrayList`	`import java.util.ArrayList;`
`LinkedList`	`import java.util.LinkedList;`
`List`	`import java.util.List;`
`Set`, `HashSet`	`import java.util.Set;` `import java.util.HashSet;`
`Map`, `HashMap`	`import java.util.Map;` `import java.util.HashMap;`
`Vector`	`import java.util.Vector;`
`Stack`	`import java.util.Stack;`
`Queue`	`import java.util.Queue;`
`Deque`, `ArrayDeque`	`import java.util.Deque;` `import java.util.ArrayDeque;`
`Iterator`	`import java.util.Iterator;`

Estructura	Importación necesaria
`Arrays` (para `Arrays.sort`, etc.)	`import java.util.Arrays;`
`Collections` (para `Collections.sort`, etc.)	`import java.util.Collections;`

✅ Resumen: `for` clásico vs `for-each` en Java

Característica	`for` clásico	`for-each` (enhanced for)
Sintaxis	`for (int i = 0; i < lista.size(); i++)`	`for (Tipo elemento : lista)`
Acceso a índice	Sí, usás `i`	No, no hay acceso directo al índice
Acceso al elemento	`lista.get(i)`	directamente `elemento`
Uso recomendado	Cuando necesitás el índice o modificar la lista	Cuando solo necesitás leer los elementos
Código más limpio	Menos legible y más propenso a errores	Más limpio y expresivo

---

🔶 Ejemplo práctico

Supongamos que tenemos esta lista:

```
java
CopiarEditar
List<String> frutas = Arrays.asList("Manzana", "Banana", "Naranja");
```

🔷 `for` clásico:

```
java
CopiarEditar
for (int i = 0; i < frutas.size(); i++) {
    System.out.println("Índice " + i + ": " + frutas.get(i));
```

```
}
```

📌 Usamos el índice `i` para acceder a cada fruta.

🔷 `for-each`:

```
java
CopiarEditar
for (String fruta : frutas) {
    System.out.println("Fruta: " + fruta);
}
```

📌 Más directo. No necesitamos saber la posición, solo queremos los elementos.

🧠 ¿Cuándo usar cada uno?

Situación	Recomendación
Necesitás el índice (`i`)	Usá `for` clásico
Recorres solo para leer datos	Usá `for-each`
Vas a modificar o eliminar elementos	Usá `for` clásico
Querés claridad y simplicidad en lectura	`for-each` es ideal
