**Java documentation Task**

1. Why does interface name ends with ‘able’?

It’s primarily used to indicate that the implementing class has the ability to perform a certain action or exhibit a certain behavior ("capability" or "functionality.").

1. Rules to declare an interface:

* Use the interface keyword.
* No method bodies (implementation) for abstract methods in an interface.
* All variables in an interface are implicitly static and final.
* Initialize the variables when you declare them because they are constants.
* All methods in an interface are public. You cannot use other access modifiers like private, protected.
* A class can implement multiple interfaces
* An interface can extend to another interface using the extends keyword.

1. Can we access the variables inside static nested class using child class

**package** com.aspiresys;

**class** Outer{

**static** **class** Inner{

**int** num =9;

}

}

**class** Child **extends** Outer{

**void** display() {

Outer.Inner inner = **new** Outer.Inner();

System.***out***.println("Child class");

System.***out***.println(inner.num);

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Child child = **new** Child();

child.display();

}

}

1. Maximum number of child class for super class

No limit but suggested to create necessary number of classes to decrease the complexity

1. Can we have class as a private?

No, A private class would be inaccessible outside.

**package** com.aspiresys;

**private** **class** Child{

**void** display() {

System.***out***.println("Child class");

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Child child = **new** Child();

child.display();

}

}

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

The constructor Child() is not visible

The type Child is not visible

**package** com.aspiresys;

**class** Child {

**private** **class** Inner {

**int** num = 9;

**public** **void** view() {

// Outer.Inner inner = new Outer.Inner();

System.***out***.println("Hi");

System.***out***.println(num);

}

}

**public** **void** display() {

Inner inner = **new** Inner();

inner.view();

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Child child = **new** Child();

child.display();

}

}

1. 15 Interfaces Names:

* Collection: The root interface for collections, like List, Set, Queue.
* Comparable: Defines a method for objects to compare themselves to others.
* Iterator: Provides methods to iterate over a collection of elements.
* Cloneable: Marks a class as capable of making a field-for-field copy of an object.
* Serializable: Marks a class whose instances can be serialized (converted to byte streams).
* AutoCloseable: Used for classes that need to release resources (like file streams)automatically when they go out of scope, especially in try-with-resources.
* Readable: Describes objects that can be read (e.g., InputStream, Reader).
* Writable: Describes objects that can be written to (e.g., OutputStream, Writer).
* Closeable: An interface for objects that need to release resources explicitly when done.
* Runnable: Represents a task that can be executed by a thread.
* Callable: Like Runnable but can return a result or throw an exception.
* Executor: Provides a way to manage and control thread execution.
* Set: A collection that does not allow duplicate elements.
* List: An ordered collection that allows duplicate elements.
* Queue: A collection used for storing elements in a specific order (FIFO).
* Deque: A double-ended queue that supports element insertion and removal from both ends.
* Map: Represents key-value pairs (e.g., HashMap, TreeMap).

1. What is throw and throws?

**throw (Used to Throw an Exception)**

The throw keyword is used to explicitly throw an exception from a method or block. When an exception is thrown, the normal flow of the program is disrupted, and the exception is passed to the nearest catch block (if available) or propagated up the call stack.

**throws (Used to Declare Exceptions in a Method Signature)**

The throws keyword is used in a method declaration to declare that a method may throw one or more exceptions. It tells the caller of the method that they need to handle the specified checked exceptions, either by using a try-catch block or by declaring them in their own method signature with throws.

1. When to use throw and throws?

* Use throw when you want to raise an exception manually based on a condition or an error in your program's logic.
* Use throws when a method might encounter an exception, and you want to inform the caller of the method that they should handle the exception (either with try-catch or by declaring it with throws in their method signature).

1. Why throw and throws?

In Scenario like if create a register form, there is a field called name . If the user submits a form without entering his/her name. Here if we use throws that may handle by taking the null value. But if we handle it explicitly using throw then we can handle it by displaying some messages to the user.

1. What is an interface?

An interface in Java is like a blueprint for creating classes. It tells us what a class should do, but it doesn’t tell us how to do it.

**Interface:**

**package** com.aspiresys.com;

**public** **interface** Readable {

// Constant

**int** ***number*** = 100;

**public** **static** String ***word*** = "Sample text";

**public** **final** **int** ***line*** = 10;

// protected long words = 62729838; //Illegal modifier for the interface field Readable.words; only public, static & final are permitted

// private Boolean flag = true;

**public** **final** **static** **long** ***totalWords*** = 34359898;

// Abstract method (no body)

**void** print();

// Default method (Java 8 onwards)

**default** **void** read() {

System.***out***.println("Reading the documents");

}

//interface will not support public protected and private methods

}

**Class:**

**package** com.aspiresys.com;

**class** Printer **implements** Readable{

@Override

**public** **void** print() {

// **TODO** Auto-generated method stub

//number=15; //The final field Readable.number cannot be assigned

System.***out***.println(***number***);

System.***out***.println("public static attribute : "+***word***);

System.***out***.println("public final attribute : "+***line***);

System.***out***.println("public final static attribute : "+***totalWords***);

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

// Create an object of Dog class

Readable printer = **new** Printer();

printer.print();

}

}

An interface is like a class, but it’s not exactly the same. It can hold constants (values that never change), method signatures (names of methods), default methods (methods with a basic implementation), static methods (methods that belong to the interface itself, not to objects).

1. What is a class?

Class is a template used to create objects. A class defines the structure and behavior that objects of that class. It is like a single unit that contains a collection of methods, variables, blocks and constructors.

1. Difference between class and interfaces?

|  |  |  |
| --- | --- | --- |
| **Feature** | **Class** | **Interface** |
| **Purpose** | Defines object structure and behavior. | Defines a contract for behavior. |
| **Methods** | Can have concrete and abstract methods. | Methods are abstract (until Java 8, default methods are allowed). |
| **Fields** | Can have instance variables and constants. | Can only have constants (public, static, final). |
| **Inheritance** | Can extend only one class. | Can extend multiple interfaces. |
| **Instantiation** | Can be instantiated with new. | Cannot be instantiated. |
| **Constructors** | Can have constructors. | Cannot have constructors. |
| **Access Modifiers** | Can have any access modifier. | Can only be public or package-private. |
| **Multiple Inheritance** | Not supported (single inheritance). | Supported (a class can implement multiple interfaces). |

1. Difference between Collections class and Collection interface?

|  |  |
| --- | --- |
| Collections class | Collection Interface |
| The Collections class is a utility class that provides static methods to operate on or return collections. It doesn't define the structure of collections but offers various utility functions. | The Collection interface is the root of the collection hierarchy in Java. It defines the fundamental operations that all collection classes must support. |
| Static methods that provide utility operations on collections. These methods don’t alter the structure of the collection but perform actions like sorting, reversing,etc...  -sort(List<T> list)  - reverse(List<T> list)  - shuffle(List<T> list) | Instance methods that are defined to be implemented by any class that implements the Collection interface include:  - add(E e)  - remove(Object o)  - clear()  - contains(Object o)  - iterator()  - size()  - isEmpty() |

1. Why is Map not included in the collection interface?

The Collection interface represents a group of individual elements. It defines basic methods for storing, accessing, and manipulating groups of objects. Map represents a key-value pair mapping. It is designed to store data in the form of associating keys with values, where each key is unique, and each key maps to a single value. Key operations in a Map are based on keys (such as put(), get(), remove(), etc.), rather than just individual elements.

1. Implementation of Priority Queue:

**package** com.aspiresys;

**import** java.util.\*;

**class** PriorityQueueCustom {

**private** **int**[] heap;

**private** **int** size;

**private** **int** capacity;

// Constructor to initialize the priority queue with a given capacity

**public** PriorityQueueCustom(**int** capacity) {

**this**.capacity = capacity;

**this**.heap = **new** **int**[capacity];

**this**.size = 0;

}

// Method to insert an element into the priority queue

**public** **void** insert(**int** element) {

**if** (size == capacity) {

**throw** **new** IllegalStateException("Priority Queue is full");

}

heap[size] = element;

size++;

heapifyUp();

}

// Method to remove and return the element with the highest priority (largest element)

**public** **int** remove() {

**if** (size == 0) {

**throw** **new** NoSuchElementException("Priority Queue is empty");

}

**int** root = heap[0];

heap[0] = heap[size - 1];

size--;

heapifyDown();

**return** root;

}

// Method to get the element with the highest priority (largest element) without removing it

**public** **int** peek() {

**if** (size == 0) {

**throw** **new** NoSuchElementException("Priority Queue is empty");

}

**return** heap[0];

}

// Method to reheapify the heap after an insert operation

**private** **void** heapifyUp() {

**int** index = size - 1;

**while** (index > 0 && heap[index] > heap[parent(index)]) {

swap(index, parent(index));

index = parent(index);

}

}

// Method to reheapify the heap after a remove operation

**private** **void** heapifyDown() {

**int** index = 0;

**while** (leftChild(index) < size) {

**int** largerChild = leftChild(index);

**if** (rightChild(index) < size && heap[rightChild(index)] > heap[largerChild]) {

largerChild = rightChild(index);

}

**if** (heap[index] >= heap[largerChild]) {

**break**;

}

swap(index, largerChild);

index = largerChild;

}

}

// Helper method to get the index of the parent of a node

**private** **int** parent(**int** index) {

**return** (index - 1) / 2;

}

// Helper method to get the index of the left child of a node

**private** **int** leftChild(**int** index) {

**return** 2 \* index + 1;

}

// Helper method to get the index of the right child of a node

**private** **int** rightChild(**int** index) {

**return** 2 \* index + 2;

}

// Helper method to swap two elements in the heap

**private** **void** swap(**int** i, **int** j) {

**int** temp = heap[i];

heap[i] = heap[j];

heap[j] = temp;

}

// Method to check if the queue is empty

**public** **boolean** isEmpty() {

**return** size == 0;

}

// Method to get the size of the priority queue

**public** **int** size() {

**return** size;

}

**public** **void** work() {

PriorityQueueCustom pq = **new** PriorityQueueCustom(10);

pq.insert(10);

pq.insert(20);

pq.insert(5);

pq.insert(30);

System.***out***.println("Priority Queue after insertions: ");

System.***out***.println("Max priority: " + pq.peek()); // Should be 30

System.***out***.println("Removed: " + pq.remove()); // Should remove 30

System.***out***.println("Max priority after removal: " + pq.peek()); // Should be 20

System.***out***.println("Removed: " + pq.remove()); // Should remove 20

System.***out***.println("Max priority after removal: " + pq.peek()); // Should be 10

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

PriorityQueueCustom obj = **new** PriorityQueueCustom(0);

obj.work();

}

}

Output:

Priority Queue after insertions:

Max priority: 30

Removed: 30

Max priority after removal: 20

Removed: 20

Max priority after removal: 10

1. Implementation of Deque / Double ended queue

**package** com.aspiresys;

**import** java.util.\*;

**class** Deque {

**private** **int**[] arr;

**private** **int** front;

**private** **int** rear;

**private** **int** size;

**private** **int** capacity;

// Constructor to initialize the deque with a given capacity

**public** Deque(**int** capacity) {

**this**.capacity = capacity;

arr = **new** **int**[capacity];

front = -1;

rear = -1;

size = 0;

}

// Method to check if the deque is full

**public** **boolean** isFull() {

**return** size == capacity;

}

// Method to check if the deque is empty

**public** **boolean** isEmpty() {

**return** size == 0;

}

// Method to insert an element at the front

**public** **void** insertFront(**int** element) {

**if** (isFull()) {

System.***out***.println("Deque is full. Cannot insert at front.");

**return**;

}

// If the deque is empty, set both front and rear to 0

**if** (isEmpty()) {

front = rear = 0;

} **else** {

front = (front - 1 + capacity) % capacity; // Circular increment

}

arr[front] = element;

size++;

}

// Method to insert an element at the rear

**public** **void** insertRear(**int** element) {

**if** (isFull()) {

System.***out***.println("Deque is full. Cannot insert at rear.");

**return**;

}

**if** (isEmpty()) {

front = rear = 0;

} **else** {

rear = (rear + 1) % capacity; // Circular increment

}

arr[rear] = element;

size++;

}

// Method to remove an element from the front

**public** **void** removeFront() {

**if** (isEmpty()) {

System.***out***.println("Deque is empty. Cannot remove from front.");

**return**;

}

**if** (front == rear) {

front = rear = -1; // Deque becomes empty

} **else** {

front = (front + 1) % capacity; // Circular increment

}

size--;

}

// Method to remove an element from the rear

**public** **void** removeRear() {

**if** (isEmpty()) {

System.***out***.println("Deque is empty. Cannot remove from rear.");

**return**;

}

**if** (front == rear) {

front = rear = -1; // Deque becomes empty

} **else** {

rear = (rear - 1 + capacity) % capacity; // Circular decrement

}

size--;

}

// Method to get the element at the front

**public** **int** getFront() {

**if** (isEmpty()) {

System.***out***.println("Deque is empty.");

**return** -1;

}

**return** arr[front];

}

// Method to get the element at the rear

**public** **int** getRear() {

**if** (isEmpty()) {

System.***out***.println("Deque is empty.");

**return** -1;

}

**return** arr[rear];

}

// Method to get the size of the deque

**public** **int** getSize() {

**return** size;

}

// Method to display the contents of the deque

**public** **void** display() {

**if** (isEmpty()) {

System.***out***.println("Deque is empty.");

**return**;

}

**int** i = front;

**while** (i != rear) {

System.***out***.print(arr[i] + " ");

i = (i + 1) % capacity;

}

System.***out***.print(arr[rear]);

System.***out***.println();

}

**public** **void** work() {

Deque deque = **new** Deque(5);

deque.insertRear(10);

deque.insertRear(20);

deque.insertFront(5);

deque.insertFront(2);

System.***out***.println("Deque after insertions:");

deque.display(); // Output: 2 5 10 20

System.***out***.println("Front element: " + deque.getFront()); // Output: 2

System.***out***.println("Rear element: " + deque.getRear()); // Output: 20

deque.removeFront();

deque.removeRear();

System.***out***.println("Deque after removals:");

deque.display(); // Output: 5 10

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

Deque obj = **new** Deque(0);

obj.work();

}

}

Output:

Deque after insertions:

2 5 10 20

Front element: 2

Rear element: 20

Deque after removals:

5 10

1. Purpose of Queue?

A Queue is typically used when you need to maintain the order of elements and process them one by one in the same order they arrived. It is most used in scenarios where you need to handle tasks sequentially, such as in scheduling or task processing systems.

1. Vector

**package** com.aspiresys;

**import** java.util.\*;

**class** SimpleIntVector {

**private** **int**[] array;

**private** **int** size;

**private** **int** capacity;

// Constructor to initialize the vector with an initial capacity

**public** SimpleIntVector(**int** initialCapacity) {

**this**.capacity = initialCapacity;

**this**.array = **new** **int**[capacity];

**this**.size = 0;

}

// Method to add an element to the vector

**public** **void** add(**int** element) {

// Check if there is enough capacity to add the element

**if** (size == capacity) {

resize();

}

array[size] = element;

size++;

}

// Method to resize the vector when capacity is reached

**private** **void** resize() {

capacity = capacity \* 2; // Double the capacity

**int**[] newArray = **new** **int**[capacity];

// Copy elements from the old array to the new array

**for** (**int** i = 0; i < size; i++) {

newArray[i] = array[i];

}

array = newArray; // Replace old array with the new one

}

// Method to get an element at a specific index

**public** **int** get(**int** index) {

**if** (index >= 0 && index < size) {

**return** array[index];

} **else** {

**throw** **new** IndexOutOfBoundsException("Index out of bounds");

}

}

// Method to remove an element at a specific index

**public** **void** remove(**int** index) {

**if** (index >= 0 && index < size) {

// Shift elements to the left to fill the gap

**for** (**int** i = index; i < size - 1; i++) {

array[i] = array[i + 1];

}

array[size - 1] = 0; // Nullify the last element

size--;

} **else** {

**throw** **new** IndexOutOfBoundsException("Index out of bounds");

}

}

// Method to return the current size of the vector

**public** **int** size() {

**return** size;

}

// Method to check if the vector is empty

**public** **boolean** isEmpty() {

**return** size == 0;

}

**public** **void** work() {

SimpleIntVector vector = **new** SimpleIntVector(2);

// Adding elements to the vector

vector.add(10);

vector.add(20);

vector.add(30); // This will trigger a resize

// Printing elements from the vector

**for** (**int** i = 0; i < vector.size(); i++) {

System.***out***.println("Element at index " + i + ": " + vector.get(i));

}

// Removing an element

vector.remove(1);

System.***out***.println("After removing element at index 1:");

**for** (**int** i = 0; i < vector.size(); i++) {

System.***out***.println("Element at index " + i + ": " + vector.get(i));

}

}

}

**class** Main {

**public** **static** **void** main(String[] args) {

SimpleIntVector obj = **new** SimpleIntVector(0);

obj.work();

}

}

Output:

Element at index 0: 10

Element at index 1: 20

Element at index 2: 30

After removing element at index 1:

Element at index 0: 10

Element at index 1: 30

1. Thread Safe

A thread-safe class or object is one where multiple threads can access and modify its data without causing inconsistent or corrupt states. A thread-safe class ensures that its methods can be safely invoked concurrently by multiple threads.

1. Dictionary in there in java?

There is no dictionary in java instead we have map.

1. Why is interface base for all class?

Interfaces provide a high degree of flexibility by decoupling the what from the how. A class that implements an interface can have different implementations of the methods.

1. Difference between HashMap, HashSet, TreeMap, TreeSet?

### **Key Differences:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **HashMap** | **HashSet** | **TreeMap** | **TreeSet** |
| **Interface** | Implements Map | Implements Set | Implements Map | Implements Set |
| **Elements** | Stores key-value pairs | Stores unique elements (no duplicates) | Stores key-value pairs, but sorted by keys | Stores unique elements, sorted in natural order or by a comparator |
| **Ordering** | No guaranteed order (unordered) | No guaranteed order (unordered) | Sorted by **key** (ascending order by default) | Sorted by **natural order** or custom comparator |
| **Null Values** | Allows **null keys** and **null values** | Allows **null elements** | **No null keys**, but allows **null values** | **No null elements** |
| **Thread Safety** | **Not synchronized** | **Not synchronized** | **Not synchronized** | **Not synchronized** |
| **Duplicates** | Allows **duplicate values**, but **no duplicate keys** | No duplicate elements allowed | No duplicate keys allowed | No duplicate elements allowed |
| **Usage** | When you need to store **key-value pairs** | When you need a **unique collection** of elements without duplicates | When you need to store **key-value pairs** sorted by key | When you need a **sorted collection** of unique elements |