```
// StackInterface.java
interface StackInterface<T> {
    void push(T item) throws StackOverflowException;
    T pop() throws StackUnderflowException;
    T peek() throws StackUnderflowException;
    boolean isEmpty();
    boolean isFull();
    int size();
}

// StackOverflowException.java
class StackOverflowException extends Exception {
    public StackOverflowException(String message) {
        super(message);
    }
}

// StackUnderflowException.java
class StackUnderflowException extends Exception {
    public StackUnderflowException(String message) {
        super(message);
    }
}


// ArrayStack.java
class ArrayStack<T> implements StackInterface<T> {
    private T[] stack;
    private int top;
    private int capacity;

    public ArrayStack() {
        this(10); // Default capacity of 10
    }

    public ArrayStack(int capacity) {
        this.capacity = capacity;
        this.stack = (T[]) new Object[capacity]; // Type erasure
warning - see explanation below
        this.top = -1;
    }

    @Override
    public void push(T item) throws StackOverflowException {
        if (isFull()) {
            throw new StackOverflowException("Stack is full. Cannot
push.");
        }
        stack[++top] = item;
    }

    @Override
    public T pop() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException("Stack is empty. Cannot
pop.");
        }
        T poppedItem = stack[top];
        stack[top--] = null; // Help with garbage collection (optional
but good practice)
        return poppedItem;
    }

    @Override
    public T peek() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException("Stack is empty. Cannot
peek.");
        }
        return stack[top];
    }

    @Override
    public boolean isEmpty() {
        return top == -1;
    }

    @Override
    public boolean isFull() {
        return top == capacity - 1;
    }

    @Override
    public int size() {
        return top + 1;
    }


    public static void main(String[] args) {
        try {
            ArrayStack<Integer> intStack = new ArrayStack<>(5);
            intStack.push(10);
            intStack.push(20);
            intStack.push(30);

            System.out.println("Popped: " + intStack.pop());
            System.out.println("Peeked: " + intStack.peek());
            System.out.println("Is empty: " + intStack.isEmpty());
            System.out.println("Is full: " + intStack.isFull());
            System.out.println("Size: " + intStack.size());


            ArrayStack<String> stringStack = new ArrayStack<>(); //
Uses default capacity
            stringStack.push("Hello");
            stringStack.push("World");
            System.out.println(stringStack.pop());


        } catch (StackOverflowException | StackUnderflowException e) {
            System.err.println("Exception: " + e.getMessage());
        }
    }
}
```

Explanation and Key Improvements:

1. **Generics:** The StackInterface and ArrayStack are now generic (<T>). This allows you to create stacks of any type (integers, strings, custom objects, etc.) without needing to rewrite the stack implementation.
2. **Exceptions:** Custom exception classes StackOverflowException and StackUnderflowException are created. This is *crucial* for proper error handling. These exceptions are used to signal when a push operation is attempted on a full stack or a pop/peek operation is attempted on an empty stack.
3. **Array Implementation:** The ArrayStack implements the StackInterface using an array to store the stack elements.
4. **Capacity:** The ArrayStack constructor now takes an optional capacity argument. This allows you to specify the initial size of the stack. A default capacity is provided if no argument is given.
5. **top index:** The top index keeps track of the top element in the stack. It's initialized to -1 for an empty stack.
6. **Push:** The push method checks for stack overflow before adding an element.
7. **Pop:** The pop method checks for stack underflow before removing an element. It also sets the popped element's array slot to null (after returning it) as a good practice to help the garbage collector.
8. **Peek:** The peek method allows you to see the top element without removing it. It also handles the underflow condition.
9. **isEmpty and isFull:** These methods provide convenient ways to check the stack's state.
10. **Size:** Returns the number of elements in the stack.
11. **Main Method (Example Usage):** The main method demonstrates how to use the ArrayStack with both Integer and String types, and how to handle the potential exceptions.
12. **Type Erasure Warning:** You'll get a warning about unchecked or unsafe operations related to the cast (T[]) new Object[capacity]. This is due to type erasure in Java generics. There's no perfect way around this with arrays and generics in Java, but this is the most

common and generally accepted approach. The warning can be suppressed with @SuppressWarnings("unchecked") if you're comfortable with the potential risks (which are minimal in this well-controlled scenario). A better, but more complex solution is to use Array.newInstance(Class<T>, int) but it adds complexity. The current solution is the most practical for a basic stack example.

This improved version provides a robust and reusable stack implementation with proper exception handling, making it much more suitable for real-world applications.