Boxing vs Unboxing

1. Boxing

- **Definition**: Converting a value type (like int, double, struct) into an object or any interface type it implements.
- **How it works**:
  - The value type is **copied** from the stack to the heap.
  - A reference to the heap object is returned.
- **Example**:

```
int num = 42;       // value type (stack)
object obj = num;    // boxing → num is copied into heap memory
```

- **Performance impact**: Creates a new object in the heap → slower and more memory-intensive than working directly with value types.

---

2. Unboxing

- **Definition**: Extracting the value type from the object or interface back into a value type variable.
- **How it works**:
  - The runtime checks if the object contains the correct value type.
  - The value is copied back from the heap to the stack.
- **Example**:

```
object obj = 42;     // boxed int
int num = (int)obj;   // unboxing → retrieves value type from heap
```

- **Performance impact**: Requires type checking and copying → also slower than working with value types directly.
- **Risks**: If the object doesn't contain the expected type, an **InvalidCastException** is thrown.

---

3. Summary Table

Feature	Boxing	Unboxing
Direction	Value type → Object	Object → Value type
Location	Stack → Heap	Heap → Stack
Performance	Slower (allocates heap memory)	Slower (type check + copy)
Exception Risk	No	Yes (InvalidCastException)

💡 **Tip**: Avoid frequent boxing/unboxing in performance-critical code. Use generics (List<int> instead of ArrayList) to keep value types unboxed.