Wrapper Classes: these are the classes in Java that encapsulate primitive data types within an object. Since primitive data types (int, double, float, char) are not objects but it is necessary to treat them as objects in certain scenarios. We can also say wrapper class provides a way to use primitive data types as reference data types (contains useful methods).

Primitive Data Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

- Working with Collections: Java collections (like ArrayList, LinkedList, etc.) can only store objects, not primitives. Wrapper classes allow you to store primitive values in collections.
- Utilizing Generic Types: Generic types in Java cannot accept primitive types as type arguments. Wrapper classes can be used as type arguments in generics.
- In order to obtain the wrapper object we need to use one of the valueOf() methods.
  e.g. Integer a = Integer.valueOf(100) . After the statement executes the value 100 is represented by an integer instance, thus 'a' wraps the value 100 within an object.
- Byte, Short, Integer, Long, Float, and Double, these are the wrapper classes that represent numeric values, and they inherit the abstract class Number.

Autoboxing & auto unboxing:

- Autoboxing is automatic conversion of a primitive type to its wrapper class whenever an object of wrapper class is needed. With the auto boxing you don't need to manually construct an object to wrap a primitive type with auto boxing you don't need to manually construct an object to wrap around a primitive type you just have to a sign the value to type wrapper reference Java automatically creates an object for you.
- `Integer iOb = 100; // autobox an int`
- similarly, auto unboxing is a process of converting wrapper class to its corresponding primitive type whenever a primitive value is required.
- `int i = iOb; // auto-unbox`

Why can't we use wrapper classes instead of primitive types?

Because each autobox and auto-unbox add overhead that is not present if the primitive type is used.

Autoboxing/ auto unboxing are not added in Java as a backdoor way of eliminating the primitive types. We only need to use wrapper classes in cases where object representation of a primitive type is required.

Advantages of using wrapper classes

Nullable Values: Wrapper classes allow you to represent nullable values by assigning null. For example, if a patient's weight or height is not available, you can set the corresponding attribute to null, which may be useful in certain scenarios.

Integration with Collections: If you need to store patient records in a collection like ArrayList or HashMap, you must use wrapper classes because these collections can only store objects, not primitives.

Disadvantages of wrapper classes - memory overhead performance overhead boxing and unboxing overhead nullity complexity potential for unintended consequences

Memory Overhead: Wrapper classes consume more memory than their corresponding primitive types. Each instance of a wrapper class has additional overhead due to the object header, reference to the object, and other metadata associated with objects in memory.

Performance Overhead: Because wrapper classes are objects, working with them can introduce performance overhead compared to primitives. Operations such as autoboxing, auto-unboxing, and method calls on wrapper objects may incur additional overhead compared to working directly with primitives.

Nullability Complexity: Wrapper classes can be null, which adds complexity when handling null values. You need to handle null checks to avoid NullPointerExceptions.

Potential for Unintended Consequences: Because wrapper classes are objects, they behave differently from primitives in certain contexts. For example, comparing two wrapper objects with == checks for reference equality, not value equality. This can lead to unexpected behavior if not handled carefully.