제네릭 타입 사용:
ReadOnly, WriteOnly, DataManager 인터페이스에 제네릭 타입을 적용하여
타입 안전성을 확보하고 불필요한 캐스팅을 줄여야 합니다.

```
interface ReadOnly<T> {
    T find(T t);
    List<T> find(Collection<T> tList); // Collection 대신 List 사용
}

interface WriteOnly<T> {
    boolean add(T t); // Object 대신 T 사용
}

interface DataManager<T> extends ReadOnly<T>, WriteOnly<T> {
    T update(T t); // Object 대신 T 사용
    boolean delete(T t);
}
```

Data 클래스의 dataList 타입 명확화:
Collection<Data> 대신 List<Data> 또는 Set<Data>와 같이 구체적인 컬렉션 타입을 사용하는 것이 좋습니다.
List는 순서가 있는 데이터, Set는 중복을 허용하지 않는 데이터에 적합합니다.

JuniorEmployee 클래스의 data 필드 활용 방안 부재:
JuniorEmployee 클래스에 Data 타입의 data 필드가 있지만, 어떻게 활용될지 명확하지 않습니다.
Data와 Employee의 관계를 명확히 하고, 필요한 메서드를 추가해야 합니다.
예를 들어, JuniorEmployee가 특정 Data를 담당하는 경우, setData()와 getData() 메서드를 추가할 수 있습니다.

MasterEmployee 클래스의 역할 명확화: MasterEmployee가
JuniorEmployee를 상속하고 DataManager 인터페이스를 구현하는 이유와 역할을 명확히 해야 합니다.
단순히 모든 기능을 가진 클래스보다는 특정 역할에 집중하는 것이 좋습니다.
예를 들어, MasterEmployee는 Data 관리에 대한 모든 권한을 가지고,
JuniorEmployee는 읽기 권한만 가지도록 설계할 수 있습니다.

인터페이스 메서드의 반환 타입:
find() 메서드의 반환 타입이 Object로 되어 있는데,
제네릭을 사용하여 타입 안전성을 확보해야 합니다.
find() 메서드가 여러 개의 객체를 반환해야 하는 경우에는 List<T>를 반환하도록 수정합니다.

예외 처리:
메서드에서 발생할 수 있는 예외를 처리하는 로직이 필요합니다.
특히 데이터베이스 관련 작업이나 파일 I/O 작업에서는 예외 처리가 필수적입니다.

ETC..:
데이터베이스 연동: 데이터를 어떻게 저장하고 관리할지 고려해야 합니다. 데이터베이스를 사용한다면,
데이터베이스 연동 로직을 추가해야 합니다.

비즈니스 로직 구현: 각 클래스의 메서드에 구체적인 비즈니스 로직을 구현해야 합니다.
테스트 코드 작성: 작성한 코드가 예상대로 동작하는지 확인하기 위해 테스트 코드를 작성하는 것이 좋습니다.

예시:

```
interface ReadOnly<T> {
    T find(T t);
    List<T> find(Collection<T> tList);
}

// ... (다른 인터페이스들)

public class JuniorEmployee<T> extends Employee implements ReadOnly<T> {
    private Data data;

    // ... (getter/setter)

    @Override
    public T find(T t) {
        // ... (구현)
    }

    @Override
    public List<T> find(Collection<T> tList) {
        // ... (구현)
    }
}
```