1. **A.**

   AbstractList:
   The ArrayList extends the AbstractList class, which provides a skeletal implementation of the List interface to minimise the effort required to implement this interface.

   Interfaces Implemented:

1. **List**:
   Represents an ordered collection (sequence) that allows duplicate elements. It provides methods for positional access and search.
2. **RandomAccess**:
   This is a marker interface used to indicate that the ArrayList supports fast (generally constant-time) random access to elements.
3. **Cloneable**:
   Indicates that the ArrayList can be cloned, enabling the use of the clone() method to create a shallow copy of the list.
4. **Serializable**:
   Indicates that the ArrayList can be serialized, which allows it to be converted into a byte stream and restored later.
5. **Collection**:
   Represents a group of objects, known as elements, and is the root interface for all collections.
6. **Iterable**:
   Allows the ArrayList to be the target of a "for-each loop," enabling iteration over its elements.

1. B.

The equals method in the Employee class was affecting the result. After overriding the equal method it's used to check the Employee object. Refer to the code for the fix

1. C.

The problem with this was that equals was overridden and there was no hashCode override method since we were using a HashMap to check if the employee is contained in the Employee HashMap. When equals is overridden, hashCode should also be overridden. Refer to the code for the fix

1. D.

This is due to a status change on the visited property. Removed visited from the equals and hashCode logic, as it should not be used for duplicate detection. Refer to the code for the fix

1. E.

When the type D is a Class:

- Java avoids the Diamond Problem by prohibiting multiple inheritance of classes.
- D must explicitly choose the desired implementation when needed. This means D extends only one class, either B or C and inherits the implementation of the extended class method(). If D needs the version of the method() from the class that it didn't inherit, it has to explicitly call the implementation within D, e.g. new C().method(); this is assuming class C wasn't inherited.

When the type D is an Interface:

- Java resolves the Diamond Problem using default methods and explicit conflict resolution via InterfaceName.super.method().
- D must override the conflicting default methods or explicitly choose one of the parent interface methods.