Option D — Object-oriented programming

A delivery company uses trains in its operations. It uses an object-oriented program to keep track of its trains and the parcels that it carries.

The company has many objects in their program; here are some of them.

Object	Description
`Train`	Each `Train` is made up of `RollingStock` objects, each of which is either a `Wagon` or an `Engine`.
`RollingStock`	A `RollingStock` object can be an `Engine` (that can pull) or a `Wagon` (that needs to be pulled). Each `RollingStock` has a unique ID number and a weight.
`Engine`	A variety of `RollingStock`. Each `Engine` has a maximum weight that it can pull.
`Wagon`	A variety of `RollingStock`. Each `Wagon` has a maximum cargo weight.
`Parcel`	Each `Parcel` is tagged with a tracking number, the addresses from where it came (origin) and to where it is going (destination) and its weight.

The code on the following pages implements the `Train` class used in this program.

(Option D continues on the following page)

(Option D continued)

```
public class Train
{
  private Engine[] mEngines;
  private Wagon[] mWagons;
  private int mEngineCount;
  private int mWagonCount;
  private int mTrainNumber;
  private double mWeight;       //  Total weight in kilograms
  public Train(int number)
  {
    mTrainNumber = number;
    mEngines = new Engine[6];  // The train can have up to 6 engines
    mEngineCount = 0;
    mWagons = new Wagon[100];  // The train can have up to 100 wagons
    mWagonCount = 0;
    mWeight = 0;
  }
  public void addEngine( Engine newEngine )
  {
    mEngines[mEngineCount] = newEngine;
    mEngineCount++;
  }
  public Engine removeEngine()
  {
    mEngineCount--;
    return mEngines[mEngineCount];
  }
  public void addWagon( Wagon newWagon )
  {
    mWagons[mWagonCount] = newWagon;
    mWagonCount++;
  }
  public Wagon removeWagon()
  {// Code to be written
  }
  public double getWeight()
  {// Code to be written
  }
  ...
}

public class RollingStock
{
  private int mIDNumber;
  private double mWeight;
  public RollingStock(int ID, double weight)
  {
    mIDNumber = ID;
    mWeight = weight;  // Weight is in kilograms
  }
  // Accessor methods
  public double getWeight() { return mWeight; }
  public int getID() { return mIDNumber; }
  ...
  // Other methods
    ...
}
```

(Option D continues on the following page)

(Option D continued)

```
public class Engine extends RollingStock
{
  private double mPullingWeight;     // maximum weight engine can pull
  public Engine(int ID)
  {
    super(ID, 120000);               // Engines weigh 120000 kilograms
    mPullingWeight = 1400000;        // Engines can pull 1400000 kilograms
  }
  // Accessor methods
  public double getWeight() { return super.getWeight(); }
  ...
  // Other methods
    ...
}

public class Wagon extends RollingStock
{
  private Parcel[] mParcels;
  private int mParcelCount;
  public Wagon(int ID)
  {
    super(ID, 32000);         // Empty wagon weighs 32000 kilograms
    mParcels = new Parcel[100];
    mParcelCount = 0;
  }
  // Accessor methods
  public int getWagonID() { return this.getID(); }
  public double getWeight()
  {
    // Code to be written
  }
  ...
  // Other methods
  ...
}
```

14. (a) Define the function of a *constructor*. *[2]*

(b) Outline the advantages of polymorphism, using the `RollingStock` class as an example. *[3]*

(c) Construct a unified modelling language (UML) diagram of the `Train` class. *[3]*

(d) Construct a method `getNumberOfWagons()`, part of the `Train` class, that returns the number of wagons currently coupled to the train. *[2]*

(e) Construct the `removeWagon()` method that will remove one wagon from a train and return the removed object. Include appropriate error checking. *[5]*

(Option D continues on the following page)

(Option D continued)

15. (a) Outline **one** advantage of using standard library collections. *[2]*

(b) Describe **two** ways in which programming by a team differs from programming by an individual working alone. *[4]*

The following code implements the `Parcel` class used in the delivery company's program.

```
public class Parcel
{
  private int trackingID;
  private double weight;
  public String destinationAddress;
  public String originAddress;
  public Parcel(int ID)
  {
    trackingID = ID;
    weight = 0;
  }
  public void setWeight(double newWeight) { weight = newWeight; }
  public double getWeight() { return weight; }
}
```

The origin and destination addresses are stored in a `Parcel` object as simple strings. However, addresses are complex and there are a lot of different pieces of information that may or may not be present such as a first name or a business name, in addition to house number, street name, city and country.

It has been decided to create a new `Address` class to handle this information.

(c) State the appropriate data type to be used in the `Address` class to store

(i) the street name; *[1]*

(ii) the building number; *[1]*

(iii) an indication of whether or not this is a business address. *[1]*

(d) Identify the changes to the `Parcel` class that will be needed to make use of the new `Address` class. *[3]*

Separate `OriginAddress` and `DestinationAddress` classes will be created. The destination address may contain special instructions to the delivery person. The origin address contains a variable that indicates if the parcel was collected from the customer's house or from the local post office.

(e) Outline how these **two** new classes can be created with minimal duplication of code. *[3]*

(Option D continues on the following page)

(Option D continued)

16. (a) Consider the following code fragment.

```
Train A = new Train(123);
Engine B = new Engine(7);
A.addEngine(B);
Wagon C = new Wagon(23);
A.addWagon(C);
Wagon D = new Wagon(66);
A.addWagon(D);
Wagon E = new Wagon(71);
A.addWagon(E);
A.addEngine(new Engine(9));
```

(i) Draw the `mEngines` array after the code fragment has been executed. *[2]*

(ii) State the value of `mEngineCount` after the code fragment has been executed. *[1]*

(iii) Draw the `mWagons` array after both the code fragment above **and** the code fragment below have been executed. *[2]*

```
Wagon F = A.removeWagon();
F = A.removeWagon();
A.addWagon(new Wagon(214));
```

The parcels loaded into a wagon cannot weigh more than the capacity of the wagon. A train's engines must have enough combined power to pull the loaded wagons. The company needs to be able to check that these requirements are being met.

(b) Construct the `getWeight()` method in the `Wagon` class that returns the total **combined** weight of the parcels currently in the wagon **and** the wagon itself. *[4]*

(c) Construct the `getWeight()` method in the `Train` class that returns the total **combined** weight of all the parcels, engines and wagons in a train. *[4]*

(d) Explain why having a `getWeight()` method in both the `Train` and `Wagon` classes does not cause a compiler error, even though the `Train` class does not inherit from the `RollingStock` class. *[2]*

(Option D continues on the following page)

(Option D continued)

17. The static array used to store Wagon objects in the Train class is to be replaced by a linked list of Wagon objects.

(a) **Without** writing detailed code, identify the changes that will need to be made. *[3]*

(b) **Without** the use of library functions, construct the addWagon() method in the Train class to add a Wagon object at the beginning of the linked list. *[3]*

(c) Describe how a method to remove a **particular** wagon would be implemented. *[5]*

The wagons have a single door so the first parcel loaded into a wagon is the last one to be unloaded.

(d) Identify a dynamic abstract data structure that would be appropriate to model the storage of parcels in a wagon. *[1]*

(e) The Wagon class has been modified to include an object named model which is an instance of the abstract data structure identified in (d). Construct code for the following methods in the Wagon class using this new object.

(i) addParcel() *[2]*

(ii) getParcel() *[2]*

(f) Explain the importance of style and naming conventions in code. *[4]*

End of Option D