Quick Computing Theory Notes (Part 3)
OOP and Linked List (in a Nutshell)

Part 1: OOP (Object-Oriented Programming)

Object-oriented programming (OOP) is a form of programming that is based on the concept of **classes** and **objects** created from the classes. These objects can **interact** by **sending messages**, **receiving messages** from other objects, and **processing data**.

This is as opposed to the conventional model, where a program consists of **functions** and **routines**.

1.1: What is a Class?

A class is the **definition** of all the **private attributes** and **public methods** which are the **common aspects** for **all objects** **created from it**. Essentially, a class acts as a **template** that have **common attributes and methods** (i.e. the same data type).

NOTE: When answering class questions not related to inheritance, use the **base class** as the example.

1.2: What is an Object?

An object is an **instance** of a class that is **created at run-time**. An object contains all the **private attributes** and **public methods** of the class it is an instance of. When an object is created, **some memory is occupied** in order for the object to hold the values for the private attributes.

How do Objects Behave?

Objects behave by **sending** and **receiving messages** from other objects, and these objects **respond accordingly** by running their **methods**.

1.3: How to Draw a Class or an Object?

Class

To draw a class: the **name** of the class, the **private attributes** and the **public methods** of the class must be drawn.

Employee	
Private:	Name EmployeeType
Public:	GetName() SetName() GetEmployeeType() SetEmployeeType() Display()

Object

To draw an object, the **name** of the object, and the **private attributes** of the object **containing values** must be drawn.

John	
Private:	Name = 'John' EmployeeType = 'H'

1.4: Inheritance

Inheritance is a situation when **a class (subclass/derived class) inherits all the attributes** and **methods** from **another class (superclass/based class)** as **part of its definition**.

Employee	
Private:	Name EmployeeType
Public:	GetName() SetName() GetEmployeeType() SetEmployeeType() Display()

SalariedEmployee	
Private:	Salary
Public:	GetSalary() SetSalary() Display()

In this example, SalariedEmployee is the **subclass** and Employee is the **superclass**. Thus, SalariedEmployee not only has the Salary attribute, it also has the Name and EmployeeType attribute from its superclass, the Employee class. Similarly, SalariedEmployee also has the GetName(), SetName(), GetEmployeeType() and SetEmployeeType() methods from its based class, Employee.

1.5: Polymorphism

Notice how SalariedEmployee also has Display() as part of its definition even though it inherits the Employee class? This is due to **polymorphism**, where different classes can **respond to the same message differently**, and is a feature of **inherited classes**. These responses are specific to the type of object.

For example,

- Display() of a **SalariedEmployee object** may display **Name**, **EmployeeType** and **Salary**, while
- Display() of an **Employee object** may display just **Name** and **EmployeeType**.

1.6: Encapsulation

Encapsulation is the mechanism in which **methods** and **attributes** are **combined** into a **single object type**.

It is the mechanism for **restricting the access** of some of the objects' components (usually **private attributes**), such that the **internal representation of an object** cannot be seen from outside of the **object's definition**. This data with restricted access typically can only be accessed by **special public methods**, known as **accessor methods (getters)** and **mutator methods (setters)**.

1.7: Support and Service Methods

Support methods are methods that **assist other methods** in performing internal tasks. They usually are **private** or **protected methods** and they cannot be called through the object.

Service methods are methods that **provide services** to the user of an object. They are **public methods** and can be accessed through the **public interface** of the object.

1.8: Issues with OOP Approach

✗ **Resource Demands**

Programs made using the object-oriented approach may require a **much greater processing overhead** than one written using traditional methods, making it work slower.

✗ **Object Persistence**

Objects that are created are **stored in the random access memory (RAM)**, **instead of** traditional methods, where data are stored in files or databases **on external storage**. Thus, there may be problems due to objects **not being able to persist** between **runs of a program**, or **between different applications.**

✗ **Reusability**

It is not easy to produce reusable objects between applications when **inheritance** is used, as it makes the class **closely coupled** with the rest of the hierarchy. **With inheritance, objects can become too application specific to reuse.** It is extremely difficult to link together different hierarchies, making it **difficult to coordinate** very large systems.

✗ **Complexity**

It is **difficult to trace and debug** the message passing between many objects in a complex application.