Chapter 3: OOP Concepts

Abstraction: Identifies the minimum essential characteristics of an entity.

- Procedural abstraction
- Data abstraction

Encapsulation: Done via classes.

- Bundling is collecting the fields (data) and methods (procedures) associated with an abstraction
- Information Hiding prevents certain aspects from being accessible by its clients. Visibility Modifiers. Private data and public accessors

Unified Modeling Language (UML) defines a graphical notation for classes

- Associations: It exists between two classes A and B if these instances can send or receive messages (make method calls) between each other.
    - Properties: Cardinality, Direction, Label (name)
    - Aggregation: Represents "**has-a**" or "**is-part-of**" relationship (open diamond)
        - Composition: Exclusive ownership is without the whole, the part can't exist. Required ownership is without the part, the whole can't exist.
    - Dependency: Represents "uses" or "knows about" relationships. It indicates coupling between classes and other relationships imply dependency (dashed arrow).
    - Associations can be unary or binary and the links are stored as private attributes.

Chapter 4: Inheritance

Inheritance: Special kind of association between classes. Creation of a hierarchy of classes, where lower-level classes share properties of a common "parent class".

- Specified with the keyword "extends" and only one is allowed per class. The absence of a parent class means that the class extends from the Object class.
- Inheritance always specifies an "**is-a**" relationship. Uses an **open triangle** in UML
- **Typical uses** of Inheritance
    - Extension defines new behavior along with keeping the existing ones.
    - Specialization modifies existing behavior(s) which can be done by overriding the parent method and modifying the method to do something that we want.
    - Specification provides implementation details of "abstract" behavior(s) which means that we provide code for the methods that the parent class has created but not given us code for.
        - Abstract classes are those that will never logically be instantiated. While abstract methods are those that cannot be completely specified at a given class level. If a class contains an abstract method, the class must be declared abstract but they can also contain concrete methods.
        - Abstract classes can declare a variable of abstract type (Vehicle v) but cannot instantiate such a variable (new Vehicle()).

Overriding: This occurs when a child class redefines an inherited method that has the same name, same parameters, and returns the same type.

- Child objects contain code for both the child method and parent method
- Calling this type of method will invoke the child method unless using super.___()

Overloading: This occurs when a child class has a method with the same name but different parameter types and can occur in the same class or parent/child classes.

- Methods with different number of parameters
- Constructors with different parameter sequences
- Different parameter type

Chapter 5: Polymorphism

Polymorphism: Means many forms

- Static polymorphism: This form of polymorphism is detectable during compilation.
    - **Operator Overloading:** An example of this can be int1 + int2 and float1 + float2 since "+" can perform on different objects. "+" is a polymorphic operation
    - **Method Overloading:** Same method name for completely different operations. This method is then considered a polymorphic method.
- Upcasting is allowed in assignments. So for example we can do Vehicle v which is the parent and we can do Airplane a = new Airplane() and then do v = a. This upcasts the airplane object as a vehicle object.
- Downcasting requires casting. So for example we can downcast the object v using a new variable called t = (Tank) v. However, this can be dangerous.

Chapter 6: Interfaces

Interface: Every class definition creates an interface which are the non-private parts of an object

- We can say that an object implements an interface. In UML we show this using a dotted line and an open triangle at the base of the interface that we are implementing.
- **Characteristics** of an Interface
    - Defines a set of methods (public)
    - Usually does not specify any implementation
    - It can have fields but they must be public, static, and final
- Subclasses inherit interface implementations and Interfaces can extend other interfaces. If a class implements an interface, it is considered a "subtype" of the "interface type" and objects can be upcast to interface types
- Apparent type: What does it look like at a particular place in the program (changes) and it determines which methods may be invoked.
- Actual type: What was it created from (never changes) and it determines which implementation to call when the method is invoked
- Abstract classes are a good choice when there is a clear inheritance hierarchy to be defined and we need non-public, non-static, or non-final fields OR private or protected methods.
- Interfaces are a good choice when the relationship between the methods and the implementing class is not extremely strong or when something like Multiple Inheritance is desired.

Chapter 7: Design Patterns

Types of Design Patterns

- Creational: Deal with the process of object creation.
    - Singleton: This is needed when we only want to have one instance of an object at a time. We provide public access to instance creation and public access to the current instance.
    - Factory Method: This is needed when we cannot anticipate which class object we must create. Therefore, it may be better to delegate specification of object types to subclasses because it is frequently desirable to avoid binding application specific classes into a set of code.
- Structural: Deal with structure of classes – how classes and objects can be combined to form larger structures. Design objects that satisfy constraints and specify connections between objects.
    - Composite: This is needed when we want objects to be organized in a hierarchical manner and where some objects are groups of other objects. Those groups along with the individuals need to be treated the same.
    - Proxy: This is needed when we have undesirable target object manipulation, expensive target object manipulation, or an inaccessible target object.
        - Protection Proxy: Controls access
        - Virtual Proxy: Acts as a stand-in
        - Remote Proxy: Local stand-in for object in another address space
- Behavioral: Deal with interaction between objects and encapsulate processes performed by objects.
    - The Iterator Pattern: This is needed when an aggregate object contains elements and the client needs access to those elements. However, the aggregate should not expose the internal structure of how those elements are stored. We can implement this using ICollection (add and getIterator) along with IIterator (hasNext, getNext) interfaces.
    - Observer: This is needed when we have an object which stores data that changes often and various clients need to use this data and be notified when it changes.

Also, when new clients are added, the code associated with the object shouldn't need to be changed.

- Observables need to be able to provide a way for observers to get registered so they can keep track of who is observing them so they can notify them when something changes.
- Observers need to say when it wants to register and provide a callback method so it can decide what to do when the data changes.

- Command: This is needed when we want to remove duplicate code that is invoked from different sources along with the desire to separate the code implementing a command from the object that invokes it.
- Strategy: This is needed when a variety of algorithms exist to perform a specific operation and the client needs to be able to select/change the algorithm at run-time.

Chapter 8: GUI Basics

Graphics Background

- Display Types
    - Cathode Ray Tube (CRT)
    - Liquid Crystal Display (LCD)
- Scan Devices
    - Random scan has arbitrary movement
    - Raster scan has a fixed raster pattern which goes to the right in the x direction and then loops back down to the next line and starts from the beginning of that line.
- Pixel is one "picture element" which consists of three color components known as R, G, and B. Screen resolution relates to pixels because it is determined by the amount of pixels that are presented on the device that is displaying the output.
- Frame Buffers occur when something like the GPU processes the commands sent from the drawing code and it writes to this buffer. The screen is then refreshed from the frame buffer.
- Double Buffering was invented to avoid flickering which would write to a back buffer and then copy from the back to the front buffer when it is done. It is important because it removes the flickering which would occur with a normal frame buffer because of the time between the draw and the clear which causes the flicker effect.

Graphics Basics

- Framework is a collection of classes that take care of low-level details of drawing objects on a screen and often provides a set of reusable screen components. These components are objects that have a graphical representation which the user usually has the ability to interact with. The framework is usually provided with an event mechanism which connects actions to the code. An example of CN1's GUI framework would be **UI**.
- Forms are a top-level container of CN1 and only one of them can be visible at a time.
- Layout Managers determine the rules for positioning the components in a container. They are classes that must be instantiated and provided to their containers.
    - Flow Layout: This layout arranges the components from left to right, top to bottom and they appear in the order that they are added. Respects preferred size.

- Border Layout: This layout adds components to five regions of the container known as North, East, South, West, and Center. These regions must be specified when a component is being added.
- Box Layout: This layout adds components to a horizontal or vertical line that doesn't break the line. Components are stretched in opposite axes.

Chapter 9: Event-Driven Programming

Event Objects

- Activating a component and use of keys and the pointer create an object of type ActionEvent. Activating a component, using a key, or use of a pointer all produce an object of type ActionEvent.

Event Listeners

- Event-driven code attaches listeners to event-generators which make call backs to the listeners. The two ways for creating one is either to have a class that implements ActionListener or a class that extends from the Command class (command pattern).

Action Listener

- If you are extending from a Component, you can override these functions (pointerPressed, pointerReleased, pointerDragged). This is the recommended approach since it is easier than adding a listener for each separate pointer action.

Chapter 10: Interactive Techniques

Component Graphics

- Every component contains an object of type Graphics which knows how to draw on the component.