Object Oriented Programming

Principles

- Basic concept: object
  - Object is a representation of an entity. It has state and behaviour
- Goals:
  - Increased understanding
  - Ease of maintenance
  - Ease of code evolution
- Main principles:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Encapsulation

- Objects encapsulate the corresponding data with the related functionality
- Rules to follow:
  - place data and the operations that perform on that data in the same class
  - use responsibility-driven design to determine the grouping of data and operations into classes

Tight Encapsulation

- Encapsulation refers to the bundling of data with the methods that operate on that data
- Tight Encapsulation emphasizes the fact that accessing fields must be only through methods
- Advantages:
  - you can monitor and validate all access and changes to the fields
  - the current data type can be hidden from users, and that gives you the opportunity later to change the type of your data without affecting the prototype of your methods and hence the other classes

Information Hiding

- information hiding is the principle of segregation of the design decisions
- the goal is to protect other parts of the program from extensive modification if the design decision is changed
- hiding data is not the full extent of information hiding
- not the same as encapsulation or tight encapsulation
- one of the key benefits of tight encapsulation is information hiding
- rules to follow:
    - do not expose data items
    - do not expose the difference between stored data and derived data
    - do not expose a class's internal structure
    - do not expose implementation details of a class

Inheritance

- inheritance is when an object or class is based on another object or class, using the same implementation
- a mechanism for code reuse
- the relationships of objects or classes through inheritance give rise to a hierarchy
- Taxomania rule:
    - every heir must introduce a feature, redeclare an inherited feature, or add an invariant clause

Polymorphism

- from Greek, meaning "having multiple forms"
- the characteristic of being able to assign a different meaning or usage to something in different contexts
- specifically, to allow an entity such as a variable, a function, or an object to have more than one form
- static polymorphism
    - overloading in Java
    - resolved during compiler time
- dynamic polymorphism
    - overriding in Java
    - resolved during run-time

Abstraction

- the process of separating ideas from specific instances of those ideas at work
- Abstract method:
    - A method that is declared but not defined (only method signature, no body)
    - declared using the abstract keyword
    - Used to put some kind of compulsion on the class who inherits the class that has absract methods. The class that inherits must provide the method implementation for the abstract method, or be declared as an abstract class
    - Constructors, static methods, private methods and final methods can't be abstract.
- Abstract class:
    - Outlines the methods but not necessarily implements all methods
    - Can't be instantitated, users needs to create another class that extends it and provides implementation of the abstract methods.
    - If a child does not implement all the abstract methods of the parent class, then it must be declared as abstract.

Converting and Casting

- Cases:
    - Conversion of primitives/objects
    - Casting of primitives/objects

Coupling

- coupling is the extent to which one object depends on another object to achieve its goal
- loose coupling is, where you minimize the dependencies an object has on other objects
- in case of tight coupling, changing the code in one class can have major effect on the dependent class, requiring changes to both classes
- a good OO design should be loosely coupled
- interfaces help achieve the loose coupling of classes

Cohesion

- cohesion is a measure of how closely related all the responsibilities, data, and methods of a class are to each other
- a good OO design should be highly cohesive

Coupling and Cohesion

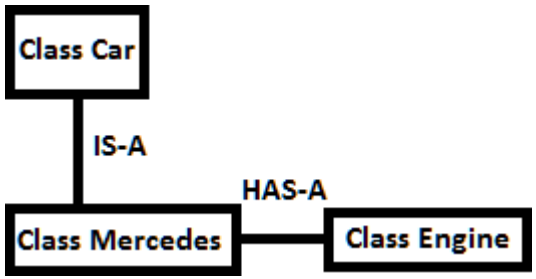
- goals of loose coupling and high cohesion:
  - ease of creation
  - ease of maintenance
  - ease of enhancement
  - less fragility
  - reusability
- implementing high cohesion works in close association with loose coupling
  - if a class is highly cohesive, it is easier to minimize the number of interactions the object has with other objects, which results in looser coupling
  - if a class performs unrelated tasks and had low cohesion, more objects will need to communicate with the class, which results in tighter coupling

Modifiers

- final
  - variable may not be changed
  - class may not be subclassed
- abstract
  - class may not be instantiated, can be subclassed
- static
  - class may not be instantiated
  - can't be overriden
- syncronized
  - prevents multiple threads accessing the same method

Object Oriented Relationships

- IS-A relationship
  - between classes and interfaces
  - extends: class inheritance
  - implements: interface implementation
- HAS-A relationship



    - based on usage, called composition
    - Class A has a reference to an instance of class B

Object Composition

- association
    - represents a relationship between two or more objects where all objects have their own lifecycle and there is no owner
- aggregation
    - variant of the "has a" association relationship
    - all object have their own lifecycle but there is ownership
    - represents “whole-part or a-part-of” relationship
    - aggregation is more specific than association
    - Example: School - Student, Team - Player
- composition
    - stronger variant of the "has a" association relationship
    - child objects do not have their lifecycle without parent object
    - composition is more specific than aggregation
    - Example: Car - Engine, Wheels; Body - Heart, Legs, Hands

Comparison of Composition and Inheritance

- Changing a class which implements composition is easier than changing a class which implements inheritance
- You can't add a method to a subclass with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
- Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
- Don't use inheritance just to get code reuse If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
- Don't use inheritance just to get at polymorphism If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

Design Pattern

- general reusable solution to a commonly occurring problem within a given context
- not a finished design that can be transformed directly into source or machine code
- description or template for how to solve a problem that can be used in many different situations

- can speed up the development process by providing tested, proven development paradigms

Singleton Pattern

- Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager.
- Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.
- It involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance
- In the same time it provides a global point of access to that instance.
- See example
- Applicability:
    - Logger classes
    - Configuration classed
    - Accessing resources in shared mode
    - Factories implemented as Singletons
- Specific implementations:
    - Thread-safe implementation for multi-threading use (a robust singleton should work when multiple threads use it)
    - Lazy instantiation using double locking mechanism
    - Early instantiation using implementation with static field

Factory Method Pattern

- Also known as Virtual Constructor, the Factory Method is related to the idea on which libraries work: a library uses abstract classes for defining and maintaining relations between objects.
- The Factory method works just the same way: it defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.
- A simple real life example of the Factory Method is the hotel:
    - When staying in a hotel you first have to check in.
    - The person working at the front desk will give you a key to your room after you've paid for the room you want and this way he can be looked at as a room factory.
    - While staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a phone-call factory, because he controls the access to calls, too.
- The pattern works with participant classes:
    - Product defines the interface for objects the factory method creates.

    - ConcreteProduct implements the Product interface.
    - Creator(also refered as Factory because it creates the Product objects) declares the method FactoryMethod, which returns a Product object. May call the generating method for creating Product objects
    - ConcreteCreator overrides the generating method for creating ConcreteProduct objects
- See example (All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Factory class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteFactory.)
- Real life example: document handlers like Excel, Word, etc. (new, open, save, close)