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