Object-Oriented Programming (OOP) Study Notes
Introduction to Object-Oriented Programming
- **Definition**: Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.
- OOP focuses on using objects that combine data and functionality.
Why OOP?
- **Modularity**: Code is organized into modules/objects.
- **Reusability**: Code can be reused across programs.
- **Scalability**: Easy to manage as systems grow.
- **Maintainability**: Changes can be made with minimal impact on existing code.

Core Concepts of OOP
1. Classes and Objects
- **Class**: A blueprint for creating objects. It defines a data structure and methods applicable to that structure.
- **Object**: An instance of a class. It can hold data and perform functions.
Example:
```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
my_car = Car("Toyota", "Corolla")
```
2. Encapsulation
- **Definition**: The bundling of data and methods that operate on data within one unit, i.e., a class.
- **Protection**: It restricts direct access to some components, which can safeguard the object's integrity.
Example:
```python
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private variable
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
```
3. Inheritance
- **Definition**: The mechanism by which one class can inherit properties from another class.
- **Benefit**: Promotes code reusability and establishes a hierarchical relationship.
Example:
```python
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return "Woof!"
my_dog = Dog()
print(my_dog.speak())  # Outputs: Woof!
```

```
### 4. Polymorphism
- **Definition**: The ability of different classes to be treated as instances of the same class through a common interface.
- **Types**:
  - **Method Overriding**: A subclass provides a specific implementation for a method that is already defined in its superclass.
  - **Method Overloading**: Multiple methods in the same scope with the same name but different parameters (not strictly supported in all languages).
**Example**:
```python
class Shape:
    def area(self):
        pass
class Rectangle(Shape):
    def area(self, length, width):
        return length * width
class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius
shapes = [Rectangle(), Circle()]
for shape in shapes:
    print(shape.area(5))  # The right area method is called based on the object type
```

Advanced Concepts of OOP
1. Abstraction
- **Definition**: The concept of hiding the complex reality while exposing only the necessary parts.
- **Implementation**: Often achieved through abstract classes and interfaces.
Example:
```python
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Square(Shape):
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side ** 2
```
2. Composition
- **Definition**: A design principle where one class contains instances of other classes as its members, establishing a "has-a" relationship.
- **Benefit**: Promotes code reuse and offers flexibility through delegation.
Example:
```python
class Engine:
    def start(self):
        return "Engine starts"
class Car:
```
```

```
    def __init__(self):
        self.engine = Engine()  # Car has an Engine
my_car = Car()
print(my_car.engine.start())  # Outputs: Engine starts
```

OOP Principles
1. DRY Principle (Don't Repeat Yourself)
- Aim to reduce repetition of code by using abstraction, inheritance, and composition.
2. SOLID Principles
- **Single Responsibility Principle**: A class should have one, and only one, reason to change.
- **Open/Closed Principle**: Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle**: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
- **Interface Segregation Principle**: No client should be forced to depend on methods it does not use.
- **Dependency Inversion Principle**: High-level modules should not depend on low-level modules, but both should depend on abstractions.

Practical Considerations
Best Practices
- Use descriptive naming for classes and methods.
- Keep classes focused on a single responsibility (SRP).
- Favor composition over inheritance when applicable.
- Document code effectively to maintain clarity.
Common Languages Supporting OOP
- **Python**: Dynamic and flexible for OOP.
- **Java**: Strongly typed with comprehensive OOP support.
- **C++**: Combines both procedural and object-oriented features.
- **C#**: Built on the Common Language Runtime (CLR) with rich OOP capabilities.

Summary
- **OOP Overview**: OOP is about organizing software design using objects that encapsulate data and functionalities.
- **Key Concepts**: Classes and objects, encapsulation, inheritance, polymorphism, abstraction, and composition.
- **Principles**: Emphasis on principles such as DRY, SOLID, and best practices to create effective and maintainable code.
- **Applicability**: Widely used in many modern languages, improving code organization and reusability.

These consolidated notes reflect foundational concepts in object-oriented programming and serve as a study guide to understand OOP principles effectively.