OOPS CHEATSHEET

Key Concepts

1. **Class**
   - **Definition**: A blueprint for creating objects that contain methods and attributes.
   - **Python**:

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
```

   - **Java**:

```
public class Animal {
    private String name;
    private String species;

    public Animal(String name, String species) {
        this.name = name;
        this.species = species;
    }
}
```

   - **C++**:

```
class Animal {
    private:
        std::string name;
        std::string species;
    public:
        Animal(std::string name, std::string species) {
```

```
            this->name = name;
            this->species = species;
        }
};
```

2. **Object**
   - **Definition**: An instance of a class.
   - **Python**:

     ```
     dog = Animal("Buddy", "Dog")
     ```

   - **Java**:

     ```
     Animal dog = new Animal("Buddy", "Dog");
     ```

   - **C++**:

     ```
     Animal dog("Buddy", "Dog");
     ```

3. **Encapsulation**
   - **Definition**: Bundling data and methods that operate on the data within one unit (class), using access specifiers.
   - **Python**:

     ```
     class Example:
         def __init__(self):
             self.__private_var = 42  # private
             self._protected_var = 84  # protected
             self.public_var = 168  # public
     ```

   - **Java**:

     ```
     public class Example {
         private int privateVar = 42;
     ```

```
    protected int protectedVar = 84;
    public int publicVar = 168;
}
```

- **C++**:

```
class Example {
    private:
        int privateVar = 42;
    protected:
        int protectedVar = 84;
    public:
        int publicVar = 168;
};
```

4. **Abstraction**
   - **Definition**: Hiding the complex implementation details and showing only the necessary features.
   - **Python**:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

class ConcreteClass(AbstractClass):
    def abstract_method(self):
        print("Implemented abstract method")
```

   - **Java**:

```
abstract class AbstractClass {
    abstract void abstractMethod();
```

```
}

class ConcreteClass extends AbstractClass {
    void abstractMethod() {
        System.out.println("Implemented abstract metho
d");
    }
}
```

- **C++**:

```
class AbstractClass {
    public:
        virtual void abstractMethod() = 0; // Pure virt
ual function
};

class ConcreteClass : public AbstractClass {
    public:
        void abstractMethod() override {
            std::cout << "Implemented abstract method"
<< std::endl;
        }
};
```


5. **Inheritance**
   - **Definition**: Forming new classes using classes that have already been defined, supporting reusability.
   - **Types**:
     - **Single Inheritance**: One class inherits from one base class.
     - **Multiple Inheritance**: One class inherits from more than one base class (not supported in Java).
     - **Multilevel Inheritance**: A class is derived from another derived class.

- **Hierarchical Inheritance**: Multiple classes inherit from one base class.
- **Hybrid Inheritance**: A combination of two or more types of inheritance.

- **Python**:

```
class Parent:
    def method1(self):
        print("Parent method")

class Child(Parent):
    def method2(self):
        print("Child method")

obj = Child()
obj.method1()
obj.method2()
```

- **Java**:

```
class Parent {
    void method1() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void method2() {
        System.out.println("Child method");
    }
}

Child obj = new Child();
obj.method1();
obj.method2();
```

- **C++**:

```
class Parent {
    public:
        void method1() {
            std::cout << "Parent method" << std::endl;
        }
};

class Child : public Parent {
    public:
        void method2() {
            std::cout << "Child method" << std::endl;
        }
};

Child obj;
obj.method1();
obj.method2();
```

6. **Polymorphism**
   - **Definition**: Presenting the same interface for different data types, achieved through method overriding and method overloading.
   - **Types**:
     - **Compile-time Polymorphism**: Achieved through method overloading and operator overloading.
     - **Runtime Polymorphism**: Achieved through method overriding.
   - **Python**:

```
class Bird:
    def sound(self):
        print("Chirp")
```

```
class Dog:
    def sound(self):
        print("Bark")

def make_sound(animal):
    animal.sound()

make_sound(Bird())
make_sound(Dog())
```

- **Java**:

```
class Bird {
    void sound() {
        System.out.println("Chirp");
    }
}

class Dog {
    void sound() {
        System.out.println("Bark");
    }
}

void makeSound(Animal animal) {
    animal.sound();
}

makeSound(new Bird());
makeSound(new Dog());
```

- **C++**:

```
class Animal {
    public:
```

```
        virtual void sound() = 0; // Pure virtual funct
ion
};

class Bird : public Animal {
    public:
        void sound() override {
            std::cout << "Chirp" << std::endl;
        }
};

class Dog : public Animal {
    public:
        void sound() override {
            std::cout << "Bark" << std::endl;
        }
};

void makeSound(Animal* animal) {
    animal->sound();
}

makeSound(new Bird());
makeSound(new Dog());
```


Principles of OOP

1. **DRY (Don't Repeat Yourself)**
    - **Definition**: Avoiding duplication by abstracting out commonalities.
    - Applies to all OOP languages.
2. **KISS (Keep It Simple, Stupid)**
    - **Definition**: Simplicity should be a key goal in design.
    - Applies to all OOP languages.

3. **SOLID Principles**
    - **Single Responsibility Principle**: A class should have only one job.
    - **Open/Closed Principle**: Classes should be open for extension but closed for modification.
    - **Liskov Substitution Principle**: Objects of a superclass should be replaceable with objects of a subclass.
    - **Interface Segregation Principle**: Many client-specific interfaces are better than one general-purpose interface.
    - **Dependency Inversion Principle**: Depend on abstractions, not concretions.

Common OOP Terms

1. **Constructor**
    - **Definition**: A special method called when an object is instantiated.
    - **Python**:

```
class Example:
    def __init__(self, value):
        self.value = value
```

    - **Java**:

```
public class Example {
    private int value;

    public Example(int value) {
        this.value = value;
    }
}
```

    - **C++**:

```
class Example {
    private:
        int value;
    public:
        Example(int value) {
            this->value = value;
        }
};
```

2. **Destructor**
   - **Definition**: A special method called when an object is destroyed.
   - **Python**:

```
class Example:
    def __del__(self):
        print("Destructor called")
```

   - **Java**:

```
// Java does not have destructors, but it has a finaliz
e method
public class Example {
    protected void finalize() {
        System.out.println("Destructor called");
    }
}
```

   - **C++**:

```
class Example {
    public:
        ~Example() {
            std::cout << "Destructor called" << std::en
dl;
```

```
        }
};
```

3. **Method Overloading**
   - **Definition**: Multiple methods with the same name but different parameters.
   - **Python**: Not natively supported but can be simulated using default arguments.

```
class Example:
    def add(self, a, b, c=0):
        return a + b + c
```

   - **Java**:

```
public class Example {
    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

   - **C++**:

```
class Example {
    public:
        int add(int a, int b) {
            return a + b
            int add(int a, int b, int c) {
             return a + b + c;
         }
 };
```

```
;
}
```

1. **Method Overriding**
   - **Definition**: Redefining a method in a subclass.
   - **Python**:

```
class Parent:
    def show(self):
        print("Parent method")

class Child(Parent):
    def show(self):
        print("Child method")
```

   - **Java**:

```
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child method");
    }
}
```

   - **C++**:

```
class Parent {
    public:
        virtual void show() {
            std::cout << "Parent method" << std::endl;
        }
};

class Child : public Parent {
    public:
        void show() override {
            std::cout << "Child method" << std::endl;
        }
};
```

2. **Super**
   - **Definition**: A function or keyword used to call a method from the parent class.
   - **Python**:

```
class Parent:
    def __init__(self, value):
        self.value = value

class Child(Parent):
    def __init__(self, value, extra_value):
        super().__init__(value)
        self.extra_value = extra_value
```

   - **Java**:

```
class Parent {
    int value;

    Parent(int value) {
```

```
        this.value = value;
    }
}

class Child extends Parent {
    int extraValue;

    Child(int value, int extraValue) {
        super(value);
        this.extraValue = extraValue;
    }
}
```

- **C++**:

```
class Parent {
    protected:
        int value;
    public:
        Parent(int value) {
            this->value = value;
        }
};

class Child : public Parent {
    private:
        int extraValue;
    public:
        Child(int value, int extraValue) : Parent(valu
e) {
            this->extraValue = extraValue;
        }
};
```


Common OOP Practices

1. **Composition over Inheritance**
   - **Definition**: Prefer composition (has-a relationship) to inheritance (is-a relationship) to achieve better modularity.
   - **Python**:

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()
```

   - **Java**:

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    Car() {
        this.engine = new Engine();
    }

    void start() {
        engine.start();
```

```
    }
}
```

- **C++**:

```
class Engine {
    public:
        void start() {
            std::cout << "Engine started" << std::endl;
        }
};

class Car {
    private:
        Engine engine;
    public:
        void start() {
            engine.start();
        }
};
```

2. **Design Patterns**
    - **Definition**: Reusable solutions to common software design problems (e.g., Singleton, Factory, Observer).
    - Applies to all OOP languages.
3. **UML Diagrams**
    - **Definition**: Visual representations of the design of a system (e.g., Class Diagrams, Sequence Diagrams).
    - Applies to all OOP languages.