ADAPTER DESIGN PATTERN OVERVIEW:

The Adapter design pattern is a structural pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, making them compatible without altering their source code. The Adapter pattern is particularly useful when integrating new code or systems with existing ones, ensuring that they can collaborate seamlessly.

Key Components of the Adapter Pattern:

1. **Target Interface:** This is the interface that the client code expects to work with. The client code is unaware of the Adapter and communicates with the Target Interface.
2. **Adaptee:** The Adaptee is the class or interface that has an incompatible interface with the Target Interface. The Adapter wraps the Adaptee and makes it compatible with the Target Interface.
3. **Adapter:** The Adapter class is responsible for implementing the Target Interface and delegating calls to the Adaptee's methods. It acts as an intermediary, translating the client's requests into a format that the Adaptee can understand.

Common Use Cases for the Adapter Pattern:

1. **Legacy Code Integration:** When you need to integrate new code or systems with existing legacy code that has a different interface, you can use the Adapter pattern to bridge the gap.
2. **Library or Framework Adaptation:** If you want to use a library or framework that doesn't directly conform to your application's interface requirements, you can create an adapter to make it compatible.
3. **Third-party API Integration:** When working with third-party APIs that have a different interface from your application's requirements, you can create adapters to make API calls and translate the responses.
4. **Polymorphism and Interface Abstraction:** Adapters can be used to enable polymorphism by allowing objects with different interfaces to be treated uniformly through a common interface.

Benefits of Using the Adapter Pattern:

1. **Flexibility:** The Adapter pattern allows for flexibility by decoupling the client code from the Adaptee. You can change or extend the Adaptee without affecting the client code.
2. **Code Reusability:** Adapters promote code reuse by enabling the use of existing classes or components in new contexts without modification.
3. **Interoperability:** It facilitates the integration of components or systems with different interfaces, ensuring they can work together harmoniously.
4. **Maintainability:** When you need to update or replace an Adaptee, you only need to modify the Adapter, reducing the risk of introducing bugs in the client code.
5. **Testing:** Adapters can simplify unit testing by allowing you to substitute mock objects for the Adaptee, making it easier to test the client code in isolation.

Conclusion:
The Adapter design pattern is a valuable tool in software development for achieving compatibility and interoperability between components with different interfaces. It promotes code flexibility, reusability, and maintainability while enabling seamless integration of new and existing code or systems. Understanding when and how to apply the Adapter pattern is essential for building robust and adaptable software solutions.

EXERCISE 1 - Classical Adapter

In this example, the Adapter class adapts the OldSystem to the NewInterface by delegating the new_method call to the legacy_method of the OldSystem.

```
# Existing interface
class OldSystem:
    def legacy_method(self):
        return "Old system is working."

# New interface expected by the client
class NewInterface:
    def new_method(self):
        pass

# Class-based Adapter
class Adapter(NewInterface):
    def __init__(self, old_system):
        self.old_system = old_system

    def new_method(self):
        return self.old_system.legacy_method()

# Using the adapter
old_system = OldSystem()
adapter = Adapter(old_system)
result = adapter.new_method()
print(result)  # Displays "Old system is working."
```

EXERCISE 2 - Object Adapter (Using Composition)
This example uses composition to adapt the LegacySystem to the NewInterface, similar to the first example.

```
# Existing interface
class LegacySystem:
    def legacy_method(self):
        return "Old system is working."

# New interface expected by the client
class NewInterface:
    def new_method(self):
        pass

# Object-based Adapter
class Adapter(NewInterface):
    def __init__(self, legacy_system):
        self.legacy_system = legacy_system

    def new_method(self):
        return self.legacy_system.legacy_method()

# Using the adapter
legacy_system = LegacySystem()
adapter = Adapter(legacy_system)
result = adapter.new_method()
print(result)  # Displays "Old system is working."
```

EXERCISE 3 - API Interface Adapter

In this case, the ApiAdapter adapts the OldApi interface to the NewApi interface, allowing the client to work with the new interface while using the existing code.

```
# Existing API interface
class OldApi:
    def request_old_data(self):
        return "Old API data"

# New API interface expected by the client
class NewApi:
    def request_new_data(self):
        pass

# API Interface Adapter
class ApiAdapter(NewApi):
    def __init__(self, old_api):
        self.old_api = old_api

    def request_new_data(self):
        return self.old_api.request_old_data()

# Using the adapter
old_api = OldApi()
adapter = ApiAdapter(old_api)
result = adapter.request_new_data()
print(result)  # Displays "Old API data."
```