- Classes
I created classes to represent different parts of a bike, such as Frame, Fork, Shock, and a main MountainBike class. Using classes helped organise the project into logical sections, made the code easier to extend later, and provided a blueprint for creating consistent components.

```python
class Suspension(Component):
      def __init__(self, name, brand, price):
         super().__init__(name, brand, price)
```

- Constructors
Each class has a constructor method (__init__) that sets up the necessary attributes like brand, price, or material. Constructors ensured that each object had all the required data from the moment it was created, helping avoid uninitialised variables and reducing bugs.

```python
def __init__(self, brand, price, travel_mm, stanchion_width):
    super().__init__("Fork", brand, price)
    self.travel_mm = int(travel_mm)
    self.stanchion_width = int(stanchion_width)
```

- Methods
Methods like total_price() and summary() in the MountainBike class perform tasks such as calculating cost and printing a build summary. This keeps logic encapsulated within objects, reducing code duplication and improving clarity.

```python
def total_price(self):
    drivetrain_total = sum(comp.price for comp in [self.chain,
self.cranks, ...])
    return drivetrain_total + ...
```

- Objects
Throughout the program, I created objects for each component (e.g. one Fork object, one Frame object) and passed them into the MountainBike object. Using objects made it easier to group data and behaviour together, and allowed different parts of the code to interact consistently.

```python
shock = Shock("Fox", 450, "Coil")
```

- Inheritance
I used inheritance to avoid repeating code. For example, Fork and Shock both inherit from Suspension, which itself inherits from Component, allowing them to automatically have attributes like brand and price without redefining them. The component class also has all of the attributes that classes like Chain need meaning they do not need any extra lines of code at all.

```python
class Chain(Component): pass
```

- Polymorphism
All components use a shared __str__ method inherited from Component, allowing the program to print or loop over different types of components without knowing their specific class. Polymorphism made it easy to handle different components in the same way, simplifying output and calculations.

```python
for extra in self.extras:
    print(f"  - {extra}")
```

- Generalisation
I generalised all parts of the bike under a common parent class Component, which stores shared values like name, brand, and price. This meant no attributes were rewritten and gave me a consistent way to store and process all parts of the bike.

Python

```python
class Component:
  def __init__(self, name, brand, price):
      self.name = name
      self.brand = brand
      self.price = float(price)

  def __str__(self):
      return f"{self.brand} {self.name} (${self.price:.2f})"

class Suspension(Component):
  def __init__(self, name, brand, price):
      super().__init__(name, brand, price)
```