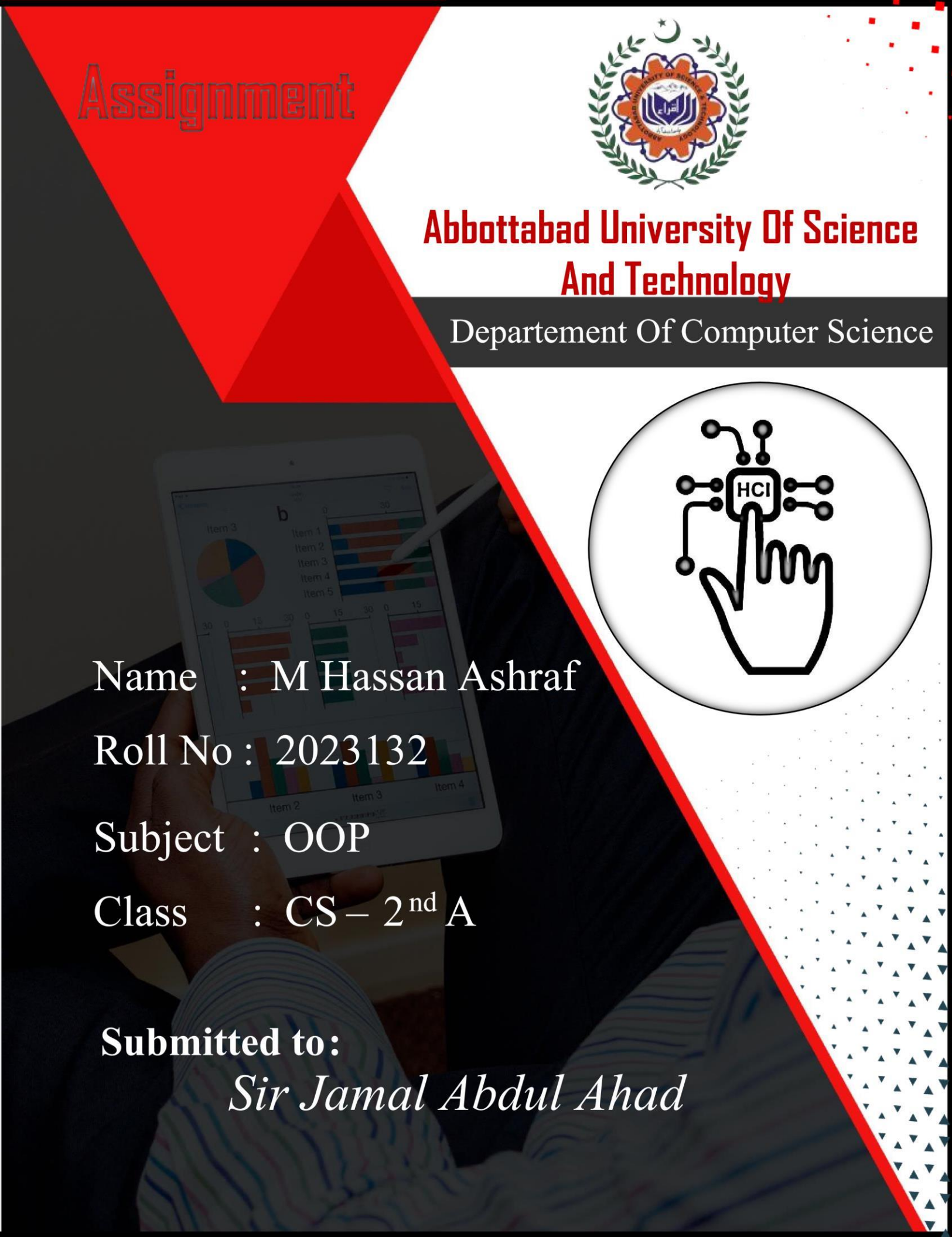
Assignment



Abbottabad University Of Science And Technology

Departement Of Computer Science

Name : M Hassan Ashraf

Roll No : 2023132

Subject : OOP

Class : CS – 2nd A

Submitted to:

Sir Jamal Abdul Ahad

Q1: Diamond Problem Resolution: Implement classes using multiple inheritance in a way that resolves the diamond problem (ambiguous method resolution).

Solution:

```
class A:
    def method(self):
        print("A method")

class B(A):
    def method(self):
        print("B method")
        super().method()  # Call A's method explicitly

class C(A):
    def method(self):
        print("C method")
        super().method()  # Call A's method explicitly

class D(B, C):  # Inherit from B, then C (order matters)
    pass

# Create an instance of D and call the method
obj = D()
obj.method()
```

Output:

```
B method
A method
C method
A method
```

Q2: Address potential pitfalls and provide illustrative examples of multiple inheritances.

Solution:

Postive Inheritance:

```
class Movable:
    def move(self):
        raise NotImplementedError("Subclasses must implement move()")

class Gripper:
    def grip(self):
        raise NotImplementedError("Subclasses must implement grip()")
    def release(self):
        raise NotImplementedError("Subclasses must implement release()")

class Robot(Movable, Gripper):
    def move(self):
        print("Robot moving...")
    def grip(self):
        print("Robot gripping...")
    def release(self):
        print("Robot releasing...")
```

Negative Inheritance:

```
class Shape:
    pass

class Drawable:
    def drawBorder(self):
        print("Drawing border...")

class Circle(Shape, Drawable):
    pass

class Square(Shape, Drawable):
    pass

class ColoredShape(Circle, Square):  # Diamond problem: Which drawBorder() to use?
    Pass
```

Alternative With Composition:

```
class ColorMixin:
    def __init__(self, color):
        self.color = color

class ColoredCircle(Circle, ColorMixin):
```

```
    pass

class ColoredSquare(Square, ColorMixin):
    pass
```

Q3: Elaborate on the nuances of employing multiple inheritance in Python.

1. **Method Resolution Order (MRO):**

```
class A:
    def method(self):
        print("A method")

class B(A):
    pass

class C(A):
    def method(self):
        print("C method")
        super().method()  # Explicitly call A's method

class D(B, C):  # Order of inheritance matters
        pass
```

2. ***Diamond Problem Resolution:***

```
class A:
    def method(self):
        print("A method")

class B(A):
    def method(self):
        print("B method")
        super().method()

class C(A):
    def method(self):
        print("C method")
        super().method()

class D(B, C):
    def method(self):
        print("D method")
```

```
        super().method()  # Calls both B and C's methods, ensuring A is invoked
```

3. Composition with Mixins:

```
class LoggerMixin:
    def log(self, message):
        print(f"Logging: {message}")

class FileSaverMixin:
    def save_to_file(self, filename):
        print(f"Saving to file: {filename}")

class DataProcessor(LoggerMixin, FileSaverMixin):
    def process_data(self, data):
        self.log("Processing data...")
        # ... processing logic ...
    self.save_to_file("processed_data.txt")
```

4. Duck Typing:

```
class Processor:
    def process(self, data):
        raise NotImplementedError

class TextProcessor:
    def process(self, data):
        print("Processing text:", data)

class ImageProcessor:
    def process(self, data):
        print("Processing image:", data)

def handle_data(processor, data):
    processor.process(data)  # No inheritance, only required method

text_processor = TextProcessor()
image_processor = ImageProcessor()
```

Q4: Illuminate the process of metaclass mechanics in Python, perhaps accompanied by an illustrative example?

```
class Metaclass(type):
    def __new__(mcs, name, bases, attrs):
```

```
        print("Creating class:", name)
        # Customize class creation here (e.g., add methods, modify attributes)
        return super().__new__(mcs, name, bases, attrs)

class MyClass(metaclass=Metaclass):
    def __init__(self, value):
        self.value = value
class AutoPropertyMetaclass(type):
    def __new__(mcs, name, bases, attrs):
        for key, value in attrs.items():
            if isinstance(value, property):
                attrs[key] = value.fget  # Extract property getter
        return super().__new__(mcs, name, bases, attrs)

class MyClass(metaclass=AutoPropertyMetaclass):
    @property
    def value(self):
        return self._value
```

2. Singleton Pattern:

```
class SingletonMetaclass(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class MySingleton(metaclass=SingletonMetaclass):

    pass
```