Abstraction

Abstraction is the process of hiding the internal implementation details and showing only the essential features to the user. It simplifies complex reality by modeling classes appropriate to the problem, and working only with the relevant information.

Why Abstraction?

Hide unnecessary details from the user

Reduce code complexity

Improve code reusability and maintainability

Focus on "what" an object does instead of "how"

Achieving Abstraction

- Abstract Classes
- Abstract Methods

We use the abc module (Abstract Base Class) to create abstract classes and methods.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

    def fuel_capacity(self):
        print("Fuel capacity varies by vehicle.")

class Car(Vehicle):
    def start(self):
        print("Car engine started with key.")

class Bike(Vehicle):
    def start(self):
        print("Bike started with self-starter.")


c = Car()
c.start()
c.fuel_capacity()

b = Bike()
b.start()
```

```
Car engine started with key.
Fuel capacity varies by vehicle.
Bike started with self-starter.
```

```
from abc import ABC, abstractmethod

# Abstract Class
class Vehicle(ABC):
    # def __init__(self, fuel):
    #     self.fuel = fuel

    @abstractmethod
    def start_engine(self):
        print('engine starts: piston moves and energy convert into combustion to work')

    # def use_fuel(self):
    #     print('fuel used : ',self.fuel)

    @abstractmethod
    def stop_engine(self):
        print('engine stops and no combustion')

# Concrete Class
class Car(Vehicle):

    def start_engine(self):
        print('engine starts')

    def car_start(self):
        print("Car starts")

    def car_stop(self):
        print("Car stops")
    def stop_engine(self):
        print('engine stops')

# Usage
# v=Vehicle('50ml')
# v.start_engine()
# c = Car('50ml')
c = Car()
# c.start_engine()
c.car_start()
c.car_stop()
# c.stop_engine()
# c.use_fuel()
```

```
Car starts
Car stops
```

Exception handling is a programming technique used to manage and respond to runtime errors or unexpected events in a graceful and controlled way, instead of letting the program crash.

Why is Exception Handling Important? Prevents the program from crashing.

- Helps in debugging.
- Allows the program to continue or safely terminate.
- Provides meaningful error messages to the user.

Exception

An exception is an error that occurs during the execution of a program.

components

- try : try Block of code to test for errors
- except : except Block of code that handles the error
- else : else Block of code to run if there is no error
- finally : finally Block of code that will always execute, error or not

- raise : raise Used to manually trigger an exception

Structure of exception handling

```
# try:
#     print('try block')
# except ZeroDivisionError:
#     print("except block")
# else:
#     print("else block')
# finally:
#     print("finally block")


#raise exception

num = -5
try:
  if num < 0:
      raise Exception("raise exception")
except Exception as e:
      print(e)
```

```
raise exception
```

```
#ZeroDivisionError
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

```
Cannot divide by zero.
```

```
#ValueError
try:
    num = int("abc")
except ValueError:
    print("Invalid literal for int conversion.")
```

```
Invalid literal for int conversion.
```

```
#IndexError
try:
    lst = [1, 2, 3]
    print(lst[5])
except IndexError:
    print("List index out of range.")
```

```
List index out of range.
```

```
#KeyError
try:
    d = {"a": 1}
    print(d["b"])
except KeyError:
    print("Key does not exist.")
```

```
Key does not exist.
```

```
#TypeError
try:
    print("abc" + 5)
except TypeError:
    print("Cannot add str and int.")
```

```
Cannot add str and int.
```

```
#AttributeError
try:
    x = 5
    x.append(10)
except AttributeError:
    print("int object has no attribute append.")
```

```
int object has no attribute append.
```

```
#FileNotFoundError
try:
    f=open("nofile.txt", "r")
except FileNotFoundError:
    print("File not found.")
```

```
File not found.
```

```
#ModuleNotFoundError
try:
    import non_existent_module
except ImportError:
    print("Module not found.")
```

```
Module not found.
```

```
#NameError
try:
    print(undeclared_var)
except NameError:
    print("Variable not defined.")
```

```
Variable not defined.
```

```
#ImportError
try:
    from math import non_existent_function
except ImportError:
    print("Import Error.")
```

```
Import Error.
```

```
try:
    print(user_name)
except NameError as e:
    print("NameError occurred:", e)
```

```
NameError occurred: name 'user_name' is not defined
```

```
person = {"name": "Alice", "age": 25}
try:
    print(person["address"])
except KeyError as e:
    print("KeyError:", e)
```

```
KeyError: 'address'
```

```
fruits = ["apple", "banana"]
try:
    print(fruits[5])
```

```
except IndexError as e:
    print("IndexError:", e)
```

```
IndexError: list index out of range
```

```
try:
    age = int("twenty")
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: invalid literal for int() with base 10: 'twenty'
```

```
try:
    result = 100 / 0
except ZeroDivisionError as e:
    print("ZeroDivisionError:", e)
```

```
ZeroDivisionError: division by zero
```

```
try:
    from math import unknown_function
except ImportError as e:
    print("ImportError:", e)
```

```
ImportError: cannot import name 'unknown_function' from 'math' (unknown location)
```

```
try:
    import non_existing_module
except ModuleNotFoundError as e:
    print("ModuleNotFoundError:", e)
```

```
ModuleNotFoundError: No module named 'non_existing_module'
```

```
try:
    with open("non_existing_file.txt", "r") as f:
        data = f.read()
except FileNotFoundError as e:
    print("FileNotFoundError:", e)
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'non_existing_file.txt'
```

```
text = "hello"
try:
    text.append(" world")
except AttributeError as e:
    print("AttributeError:", e)
```

```
AttributeError: 'str' object has no attribute 'append'
```

```
try:
    result = "5" + 5
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: can only concatenate str (not "int") to str
```

```
def withdraw(amount):
    if amount < 0:
        raise ValueError("Amount cannot be negative")
try:
    withdraw(-100)
except ValueError as e:
    print("Custom Exception:", e)
```

```
Custom Exception: Amount cannot be negative
```

```
try:
    with open("filename.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print(f"File not found!")
```

```
File not found!
```

```
config = {"host": "localhost"}
try:
    print(config["port"])
except KeyError:
    print("Missing configuration: 'port'")
```

```
Missing configuration: 'port'
```

```
orders = ["Pizza", "Burger"]
try:
    print("Next order:", orders[5])
except IndexError:
    print("Order queue is empty or invalid index")
```

```
Order queue is empty or invalid index
```

```
def multiply(a, b):
    return a * b

try:
    print(multiply(5, "hello", 2))
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: multiply() takes 2 positional arguments but 3 were given
```

```
response = None
try:
    print(response.upper())
except AttributeError as e:
    print("AttributeError:", e)
```

```
AttributeError: 'NoneType' object has no attribute 'upper'
```

```
class InsufficientFunds(Exception):
    pass
balance = 500
withdraw_amount = 1000

try:
    if withdraw_amount > balance:
        raise InsufficientFunds("Not enough balance!")
except InsufficientFunds as e:
    print(e)
```

```
Not enough balance!
```

Start coding or generate with AI.