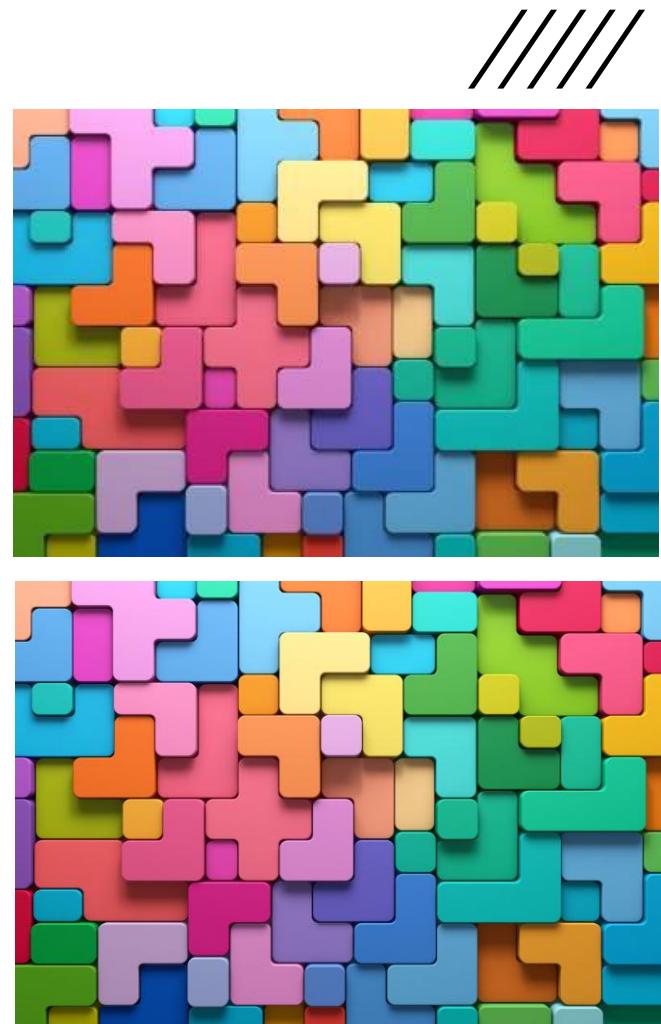


# A R V

FORELESNING 9

MANDAG 15/9

(bilder generert av bing image creator)





# Læremål: Avansert bruk av Python

- Gjøre programmene lettere å lese og finne fram i
- *Objektorientert programmering* er en måte å tenke på når vi skriver programmer som åpner nye muligheter
- Lar oss lage spesialtilfeller fra mer generelle tilfeller (arv)
- (+ mer de kommende ukene)

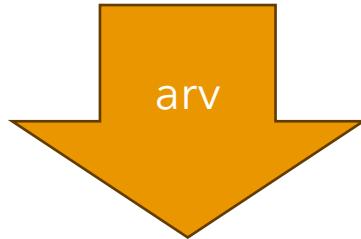


# • Klassediagram

Generelle  
klasser



arv



Mer  
spesialiserte  
klasser

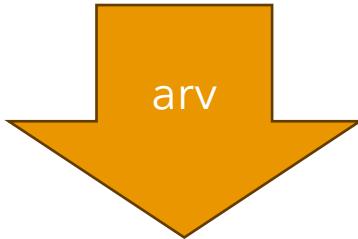


inneholder



Kort

arv



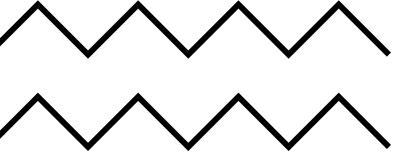
KlassiskKort



# • Bør vi alltid bruke arv?

- Nei, ikke alltid!
- Arv *kan* være fornuftig hvis
  - Vi vil ha en ny klasse som gjør mye av det en annen klasse gjør + noe ekstra (slipper å copy-paste kode)
  - Du har flere klasser som gjør akkurat det samme, og vil oppdatere dem *ett sted* (i moderklassen) i stedet for å oppdatere dem hver for seg (**Goose** er oppdatert, men du glemte **Duck** og **Swan**)
  - Du bruker type-sjekking og synes **b: Bird** er mer oversiktelig enn **b: Duck | Goose | Albatross**

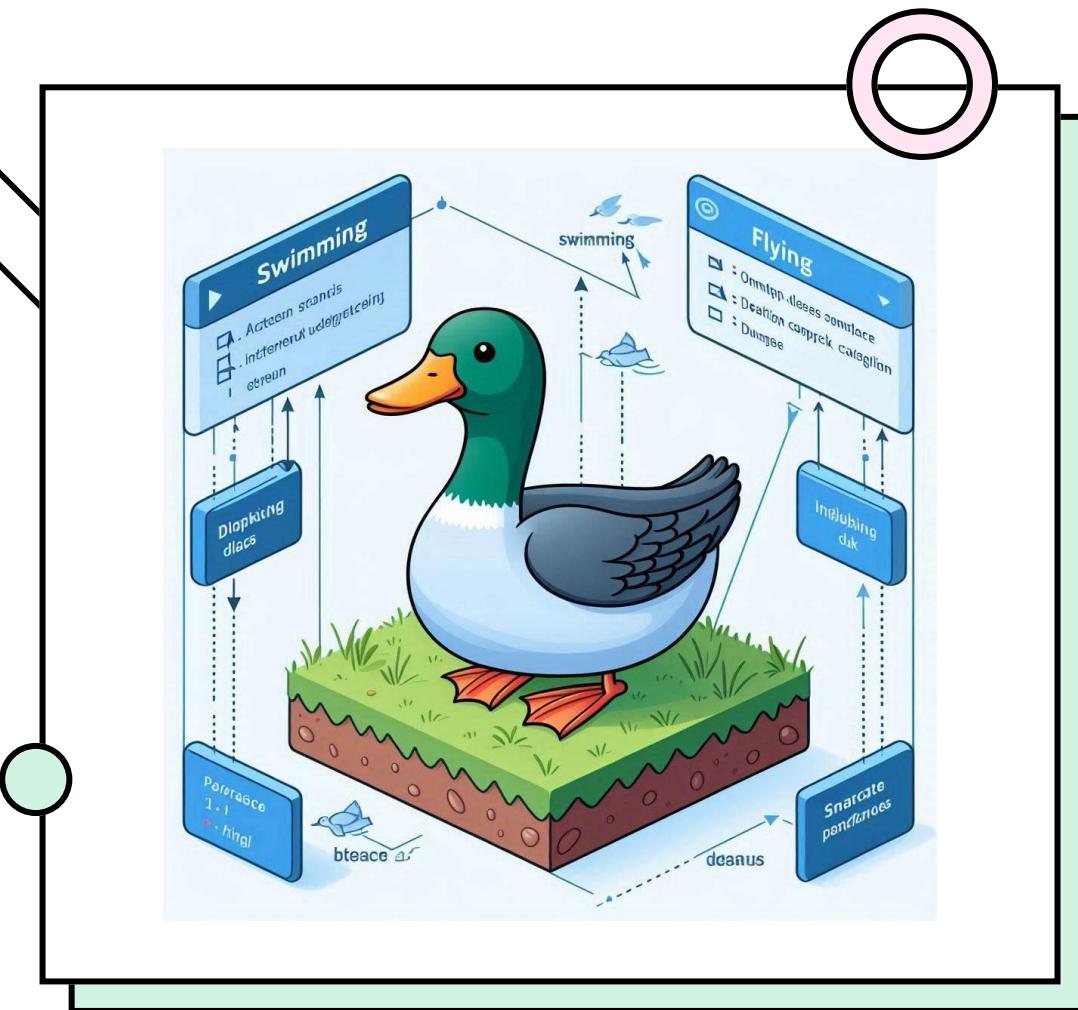




# Hva er alternativet?

- *Duck typing* (Python):
  - “if it `.swim()` like a **Duck** and `.fly()` like a **Duck**, it must be a **Duck**”
  - alle klasser med de rette metodene gjør at koden kjører uten feil, selv om det ikke er “riktig” klasse
  - (men **mypy** gjennomskuer det med en gang)





# LIVE KODING :

# D U C K T Y P I N G O G

# I M P E L E M E T N E R E

# G R E N S E S N I T T

# ( A B S T R A K T E

# K L A S S E R )





```
class Duck:
    def swim(self) -> None:
        print("The duck is swimming.")

    def fly(self) -> None:
        print("The duck is flying.")

class Goose:
    def swim(self) -> None:
        print("The goose is swimming.")

    def fly(self) -> None:
        print("The goose is flying.")

class Albatross:
    def swim(self) -> None:
        print("The albatross is swimming.")

    def fly(self) -> None:
        print("The albatross is flying.")
```

# Duck typing (uten arv)

```
birds:list[Duck] = [Duck(), Goose(), Albatross()]

for bird in birds:
    bird.fly()
    bird.swim()
```

```
The duck is flying.
The duck is swimming.
The goose is flying.
The goose is swimming.
The albatross is flying.
The albatross is swimming.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 5> mypy ducks.py
ducks.py:23: error: List item 1 has incompatible type "Goose"; expected "Duck"  [list-item]
ducks.py:23: error: List item 2 has incompatible type "Albatross"; expected "Duck"  [list-item]
Found 2 errors in 1 file (checked 1 source file)
```





# Samme eksempel med arv

```
class Bird:  
    def swim(self) -> None:  
        print(f"The {type(self).__name__.lower()} is swimming.")  
  
    def fly(self) -> None:  
        print(f"The {type(self).__name__.lower()} is flying.")
```

```
class Duck(Bird):  
    pass  
  
class Goose(Bird):  
    pass  
  
class Albatross(Bird):  
    pass
```

```
birds: list[Bird] = [Duck(), Goose(), Albatross()]  
  
for bird in birds:  
    bird.fly()  
    bird.swim()
```

```
The duck is flying.  
The duck is swimming.  
The goose is flying.  
The goose is swimming.  
The albatross is flying.  
The albatross is swimming.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 5> mypy birds.py  
Success: no issues found in 1 source file
```



```
from abc import ABC, abstractmethod

class Swimming(ABC):
    @abstractmethod
    def swim(self) -> None:
        raise NotImplementedError("swim() must be implemented!")

class Flying(ABC):
    @abstractmethod
    def fly(self) -> None:
        raise NotImplementedError("fly() must be implemented!")
```

# Tredje mulighet: Implementere grensesnitt (abstrakte klasser)

```
class Duck(Swimming, Flying):
    def swim(self) -> None:
        print("The duck is swimming.")

    def fly(self) -> None:
        print("The duck is flying.")

class Penguin(Swimming):
    def swim(self) -> None:
        print("The penguin is swimming.")

class Frigatebird(Flying):
    def fly(self) -> None:
        print("The frigatebird is flying.")
```

```
birds: list[object] = [Duck(), Penguin(), Frigatebird(), "not_a_bird"]

for bird in birds:
    if isinstance(bird, Flying):
        bird.fly()
    if isinstance(bird, Swimming):
        bird.swim()
```

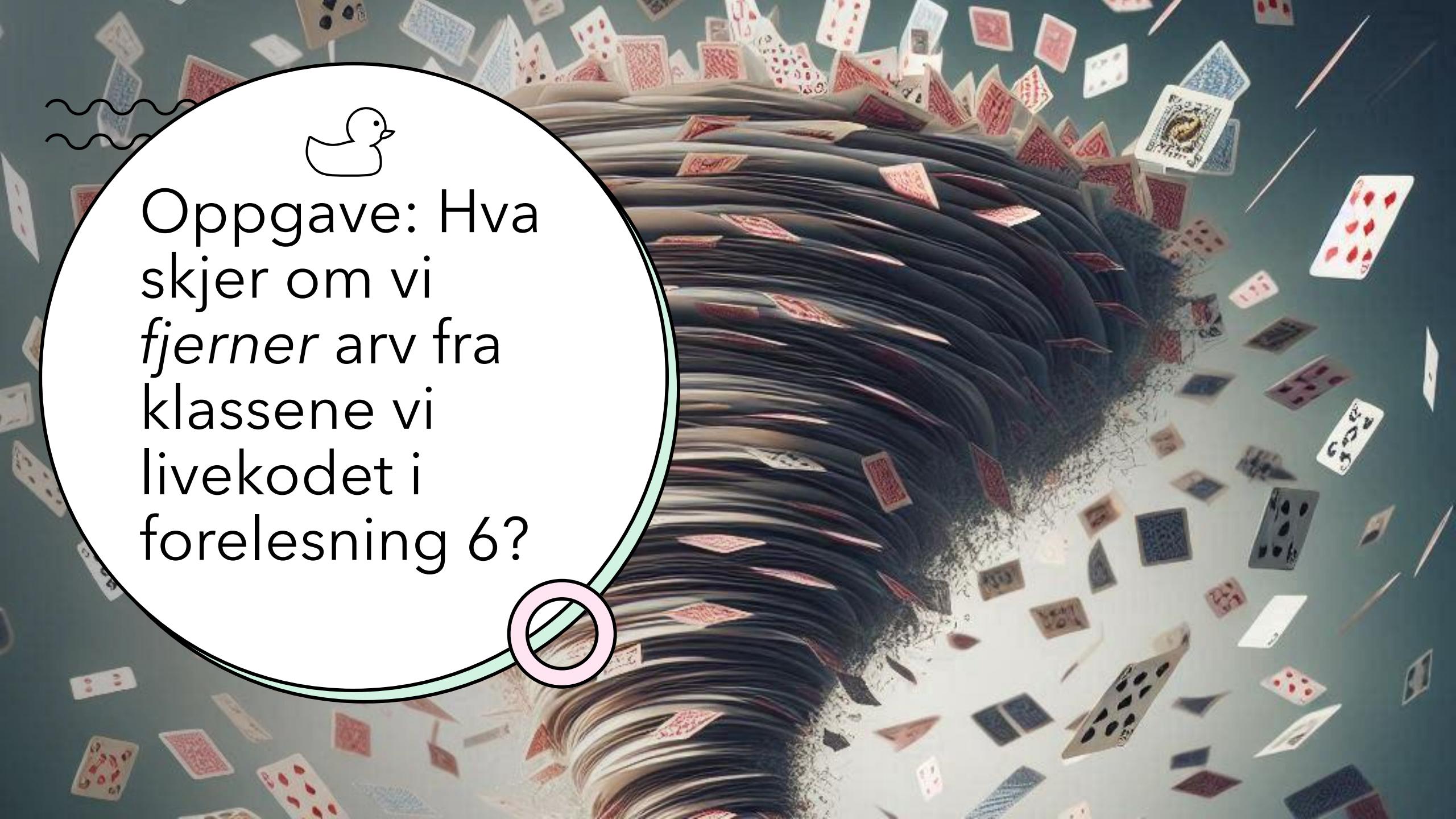
```
The duck is flying.
The duck is swimming.
The penguin is swimming.
The frigatebird is flying.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 7> mypy abstract.py
Success: no issues found in 1 source file
```





Oppgave: Hva skjer om vi *fjerner* arv fra klassene vi livekodet i forelesning 6?



# ○ Fordeler med duck typing i Python

- Koden blir enkel og fleksibel når du kan fokusere på hva objektene gjør og ikke hva de er
- Du kan lage enkle *prototyper* av mer kompiserte klasser for å teste annen kode i en tidlig fase
- Du kan gjenbruke en klasse et annet sted uten å importere alle klassene som arver fra hverandre



# Ulemper med duck typing i Python

- Programmet kan kræsje eller gi feilmelding når det kjøres fordi det mangler en metode eller attributt som man antok var der
- Det kan være vanskelig å vite hva slags objekt det er snakk om når man leser koden
- Det kan bli mer krevende å finne feil når typene er udefinert

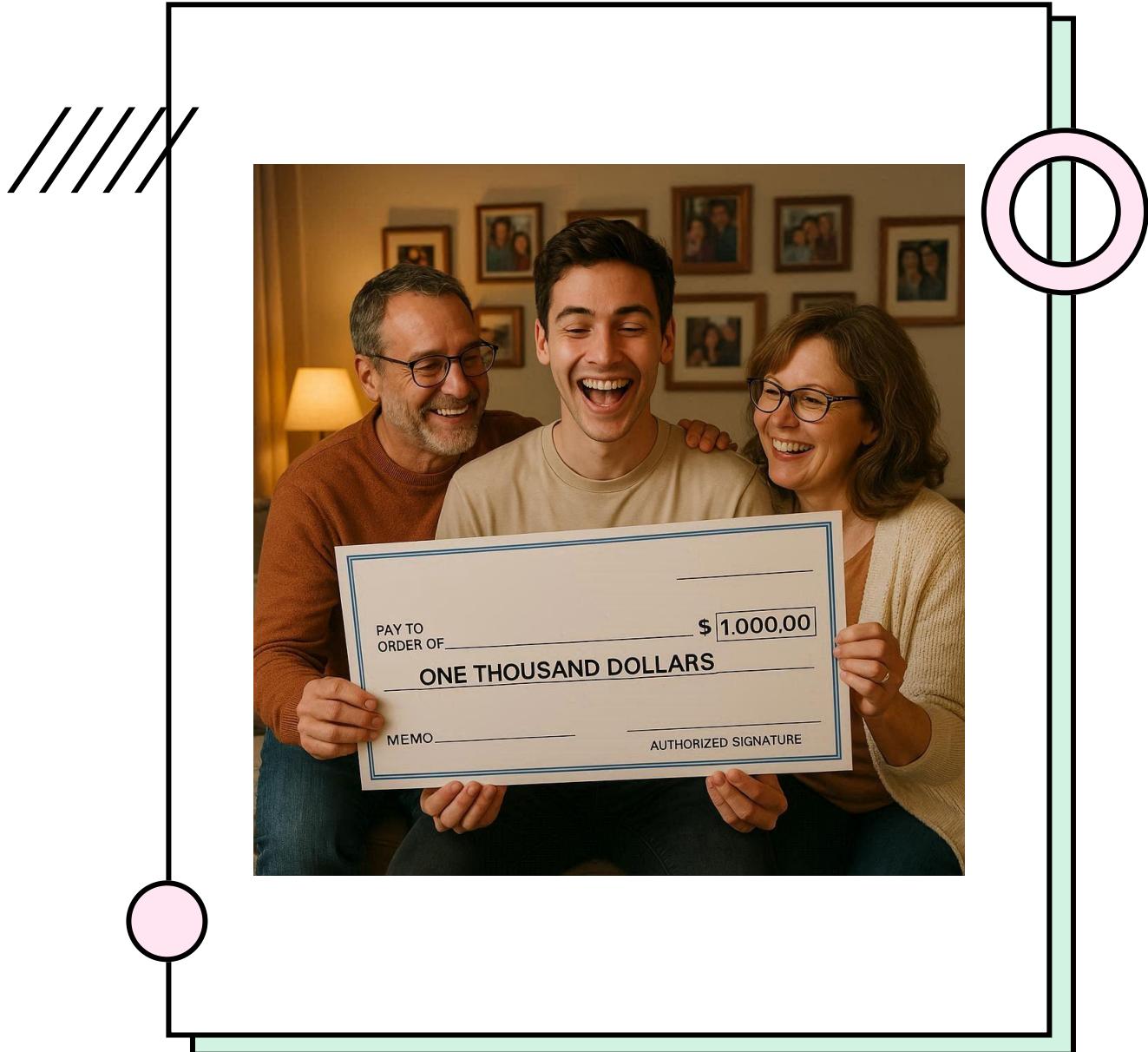


- Definere grensesnitt (interface) med abstract base classes (ABCs)

- En annen måte å bruke arv på
- Lager en *abstrakt klasse* = en du ikke kan lage objekter av (du kan ikke lage en **Swimming**, men du kan lage en **Duck**)
- En klasse som *arver* fra denne klassen må *implementere* alle metodene i grensesnittet
- Hvis du glemmer en får du **NotImplementedError**
- Kan dermed *tvinge* et objekt til å ha bestemte metoder fra ett (eller flere) grensesnitt



# LIVEKODING : ARVE FRA FLERE KLASSER



- Python støtter generelt arv fra flere klasser samtidig

```
class Stipendiat(Arbeidstaker, Student):  
    def __init__(self):  
        Arbeidstaker.__init__(self)  
        Student.__init__(self)
```

← Her kan vi ikke bruke **super()** men må gi beskjed om hvilken moderklasse vi kaller ting fra!

