# Mini Project: Vikings Presentation

Team Baby Spice
Desi, Iva, Jan Dirk
Date: 09.05.2025

## Mini project - Vikings

### Problem

→ Simulate war between Vikings and Saxons.
→ OOP concepts: inheritance, method overriding, class hierarchy.

### Solution

→ Class Design
→ Testing
→ Battle Simulation

### Learnings

→ Live Share
→ Inheritance and override
→ Fixing errors

### Conclusion

→ New coding strategies
→ Cleaner code
→ New syntax

## Battle Between Viking and Saxons

## The problem: Learning OOP

Simulate battles between Vikings and Saxons using object-oriented programming in Python.

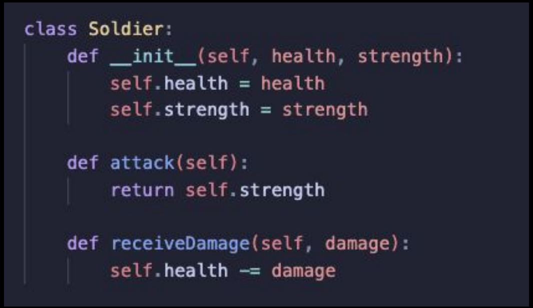
The Challenge:

→ Build a class hierarchy: Soldier, Viking, Saxon, and War.

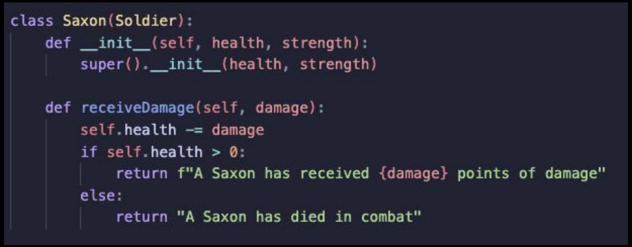
→ Implement combat behavior using inheritance and method overrides.

→ Manage armies and simulate attacks between units.

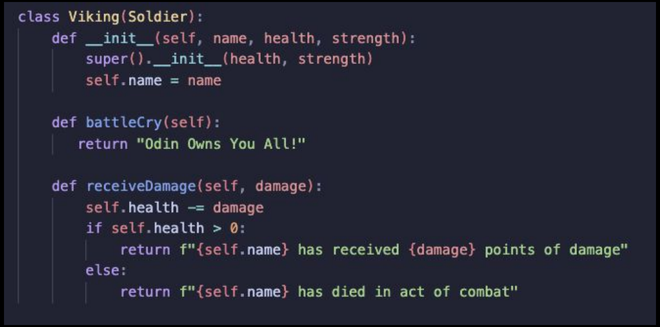
# Solution
Soldiers



```python
class Soldier:
    def __init__(self, health, strength):
        self.health = health
        self.strength = strength

    def attack(self):
        return self.strength

    def receiveDamage(self, damage):
        self.health -= damage
```



```python
class Saxon(Soldier):
    def __init__(self, health, strength):
        super().__init__(health, strength)

    def receiveDamage(self, damage):
        self.health -= damage
        if self.health > 0:
            return f"A Saxon has received {damage} points of damage"
        else:
            return "A Saxon has died in combat"
```



```python
class Viking(Soldier):
    def __init__(self, name, health, strength):
        super().__init__(health, strength)
        self.name = name

    def battleCry(self):
       return "Odin Owns You All!"

    def receiveDamage(self, damage):
        self.health -= damage
        if self.health > 0:
            return f"{self.name} has received {damage} points of damage"
        else:
            return f"{self.name} has died in act of combat"
```

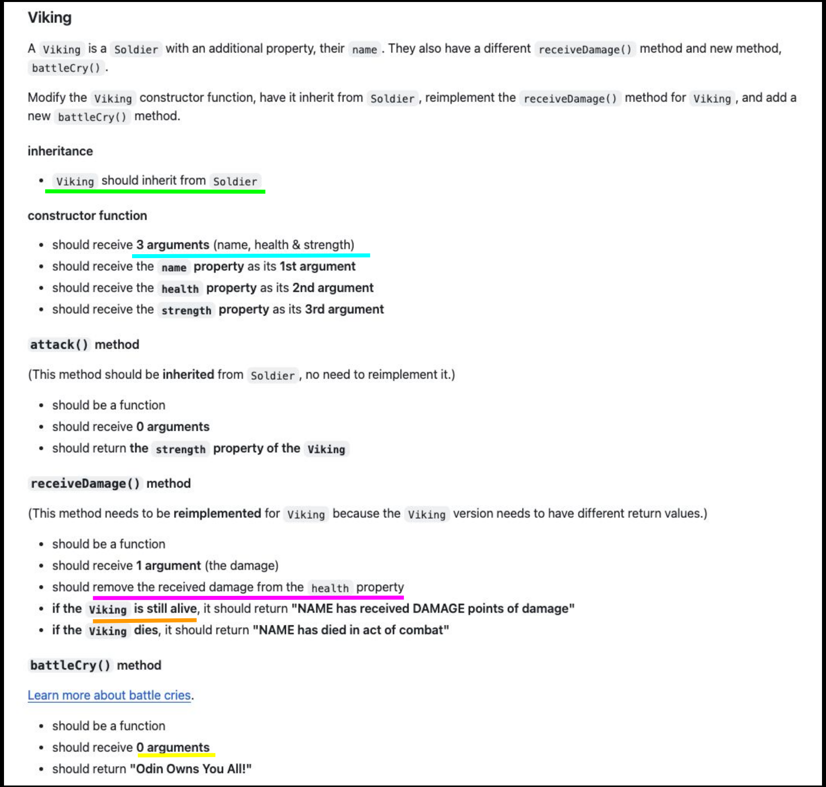
→ Created base class "Soldier" and derived classes "Saxon" and "Viking"

→ Used super() to call base class "Soldier"

→ Adjusted the receiveDamage function to each derived class.

→ Conditional statements to check whether soldier is damaged or died.

# Vikings



### Viking

A `Viking` is a `Soldier` with an additional property, their `name`. They also have a different `receiveDamage()` method and new method, `battleCry()`.

Modify the `Viking` constructor function, have it inherit from `Soldier`, reimplement the `receiveDamage()` method for `Viking`, and add a new `battleCry()` method.

#### inheritance

- `Viking` should inherit from `Soldier`

#### constructor function

- should receive **3 arguments** (name, health & strength)
- should receive the **`name` property** as its **1st argument**
- should receive the **`health` property** as its **2nd argument**
- should receive the **`strength` property** as its **3rd argument**

#### `attack()` method

(This method should be **inherited** from `Soldier`, no need to reimplement it.)

- should be a function
- should receive **0 arguments**
- should return **the `strength` property of the `Viking`**

#### `receiveDamage()` method

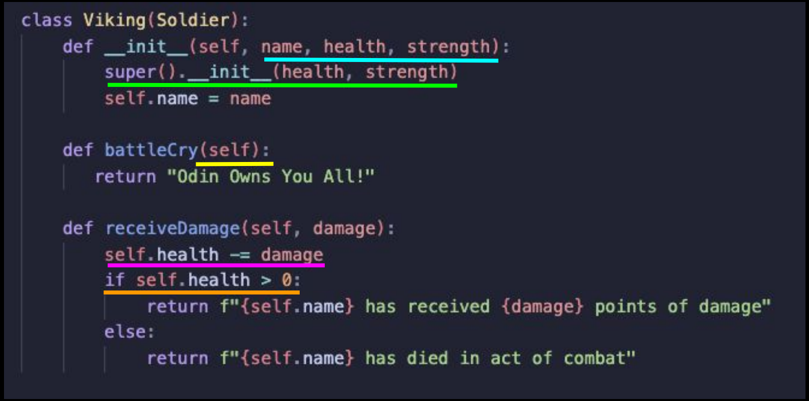
(This method needs to be **reimplemented** for `Viking` because the `Viking` version needs to have different return values.)

- should be a function
- should receive **1 argument** (the damage)
- should remove the received damage from the `health` property
- **if the `Viking` is still alive**, it should return **"NAME has received DAMAGE points of damage"**
- **if the `Viking` dies**, it should return **"NAME has died in act of combat"**

#### `battleCry()` method

Learn more about battle cries.

- should be a function
- should receive **0 arguments**
- should return **"Odin Owns You All!"**



```python
class Viking(Soldier):
    def __init__(self, name, health, strength):
        super().__init__(health, strength)
        self.name = name

    def battleCry(self):
        return "Odin Owns You All!"

    def receiveDamage(self, damage):
        self.health -= damage
        if self.health > 0:
            return f"{self.name} has received {damage} points of damage"
        else:
            return f"{self.name} has died in act of combat"
```

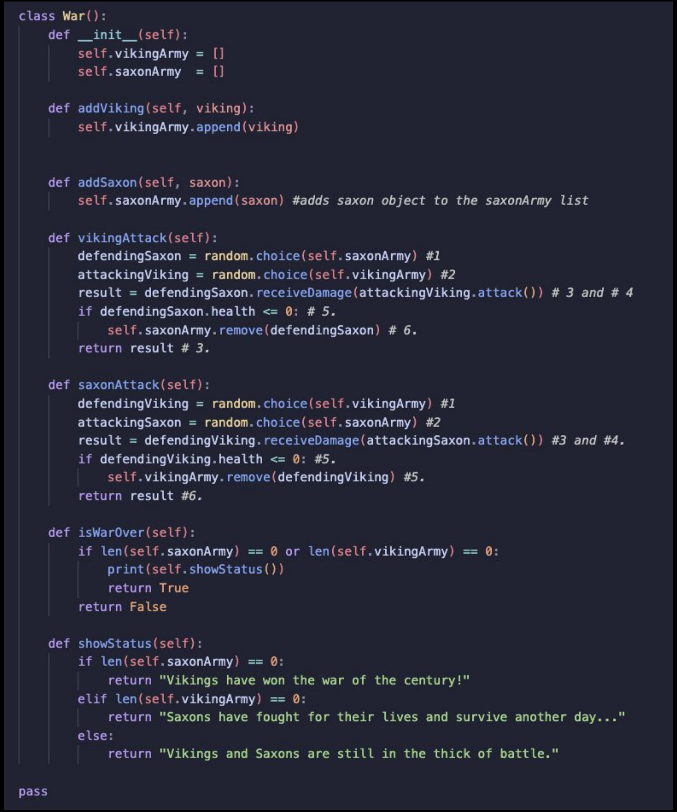
## Solution
## War

→ Purpose: Implement War Class
- Build armies, handle attacks, track health, removes dead soldiers, ends war.

→ Initialize empty armies

→ Battle logic:
- Select attacker and defender
- Calculate and apply damage
- Remove dead soldiers
- Check if enough soldiers to start a battle

→ Game Control:
- Return true if either army is empty
- Return a message based on wins, survival and battle status



```python
class War():
    def __init__(self):
        self.vikingArmy = []
        self.saxonArmy  = []

    def addViking(self, viking):
        self.vikingArmy.append(viking)


    def addSaxon(self, saxon):
        self.saxonArmy.append(saxon) #adds saxon object to the saxonArmy list

    def vikingAttack(self):
        defendingSaxon = random.choice(self.saxonArmy) #1
        attackingViking = random.choice(self.vikingArmy) #2
        result = defendingSaxon.receiveDamage(attackingViking.attack()) # 3 and # 4
        if defendingSaxon.health <= 0: # 5.
            self.saxonArmy.remove(defendingSaxon) # 6.
        return result # 3.

    def saxonAttack(self):
        defendingViking = random.choice(self.vikingArmy) #1
        attackingSaxon = random.choice(self.saxonArmy) #2
        result = defendingViking.receiveDamage(attackingSaxon.attack()) #3 and #4.
        if defendingViking.health <= 0: #5.
            self.vikingArmy.remove(defendingViking) #5.
        return result #6.

    def isWarOver(self):
        if len(self.saxonArmy) == 0 or len(self.vikingArmy) == 0:
            print(self.showStatus())
            return True
        return False

    def showStatus(self):
        if len(self.saxonArmy) == 0:
            return "Vikings have won the war of the century!"
        elif len(self.vikingArmy) == 0:
            return "Saxons have fought for their lives and survive another day..."
        else:
            return "Vikings and Saxons are still in the thick of battle."

pass
```

# Learnings & challenges

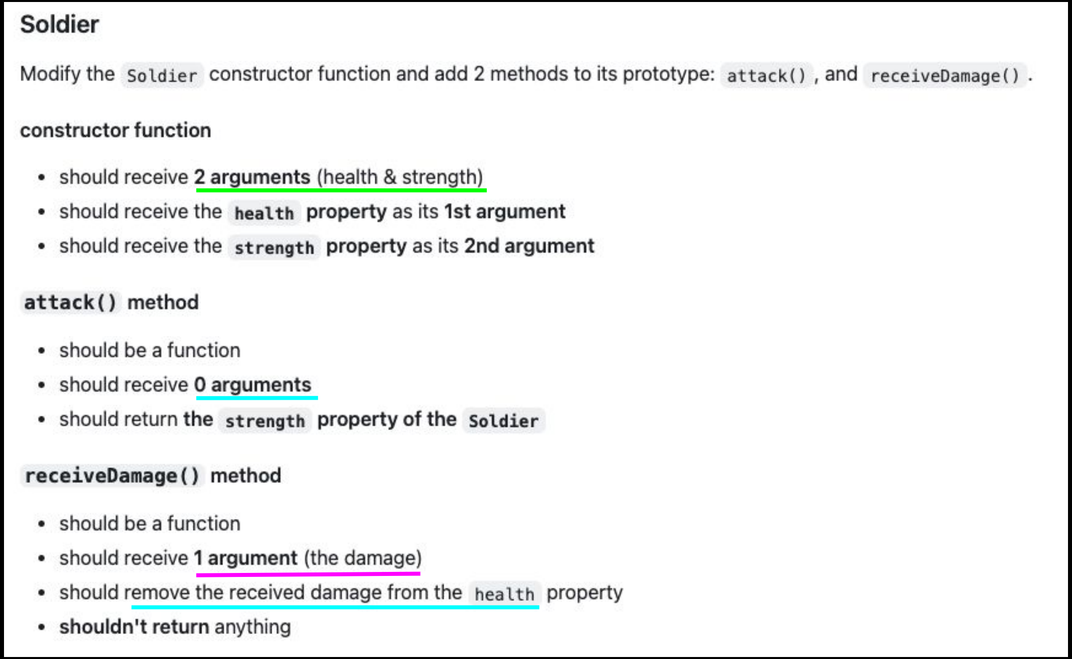
- → Learned how to apply inheritance and override methods to reflect unique behavior
- → Learned how to use Live Share on VSCode
- → Error when there wasn't enough soldiers → created IsWarOver function to first check if there's enough soldiers on both sides to initiate a battle.
- → Learned how to build and connect multiple classes into a functional system.
- → Challenge: Tracking nameless Saxon class

## Conclusion

- → Translate instructions into pseudocode helps to structure the solution.
- → Live Share is cool, but it is messy to run the tests.
- → Python is a different syntax but we are getting used to it.
- → Designing clean, reusable code takes more work but it pays out in the end.

Thanks :)
questions?

# Solution
Soldiers



### Soldier

Modify the `Soldier` constructor function and add 2 methods to its prototype: `attack()`, and `receiveDamage()`.

**constructor function**

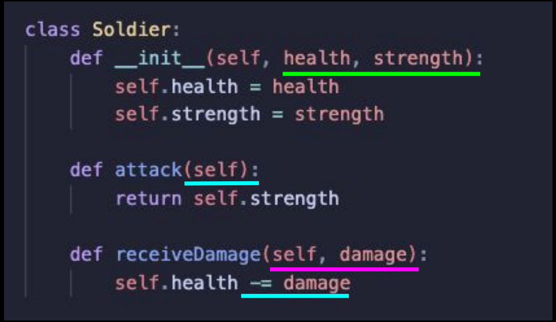
- should receive **2 arguments** (health & strength)
- should receive the `health` **property** as its **1st argument**
- should receive the `strength` **property** as its **2nd argument**

**`attack()` method**

- should be a function
- should receive **0 arguments**
- should return **the** `strength` **property of the** `Soldier`

**`receiveDamage()` method**

- should be a function
- should receive **1 argument** (the damage)
- should remove the received damage from the `health` property
- **shouldn't return** anything



```python
class Soldier:
    def __init__(self, health, strength):
        self.health = health
        self.strength = strength

    def attack(self):
        return self.strength

    def receiveDamage(self, damage):
        self.health -= damage
```