

Class vs. instance attributes

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON



George Boorman
Curriculum Manager, DataCamp

Core principles of OOP

Encapsulation:

- Bundling of data and methods

Inheritance:

- Extending the functionality of existing code

Polymorphism:

- Creating a unified interface

Instance-level attributes

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

emp1 = Employee("Teo Mille", 50000)
emp2 = Employee("Marta Popov", 65000)
```

- `name` and `salary` values are specific to each object
- `self` assigns to an object

Class-level attributes

- Data shared among all instances of a class
- Define *class attributes* in the body of `class`
- "Global variable" within the class

Implementing class-level attributes

```
class Employee:
    # Define a class attribute
    # No self. syntax
    MIN_SALARY = 30000
    def __init__(self, name, salary):
        self.name = name
        # Use class name
        # to access class attribute
        if salary >= Employee.MIN_SALARY:
            self.salary = salary
        else:
            self.salary = Employee.MIN_SALARY
```

- `MIN_SALARY` is shared among all instances
- Don't use `self` to *define* class attribute
- Convention is to use capital letters
- Use `ClassName.ATTR_NAME` to *access* the class attribute value

Class-level attributes

```
class Employee:
    # Define a class attribute
    MIN_SALARY = 30000
    def __init__(self, name, salary):
        self.name = name
        # Use class name
        # to access class attribute
        if salary >= Employee.MIN_SALARY:
            self.salary = salary
        else:
            self.salary = Employee.MIN_SALARY
```

```
emp1 = Employee("John", 40000)
print(emp1.MIN_SALARY)
```

30000

```
emp2 = Employee("Jane", 60000)
print(emp2.MIN_SALARY)
```

30000

Modifying class-level attributes

```
emp1 = Employee("John", 40000)
emp2 = Employee("Jane", 60000)

# Update MIN_SALARY of emp1
emp1.MIN_SALARY = 50000

# Print MIN_SALARY for both employees
print(emp1.MIN_SALARY)
print(emp2.MIN_SALARY)
```

```
50000
```

```
30000
```

Modifying class-level attributes

- `MIN_SALARY` is created in the class definition
- Updating `MIN_SALARY` of an object will not affect the value in the class definition
- Security - prevent changes being made to software!

Why use class attributes?

Global constants related to the class

- Minimum and maximum values for attributes
 - Prevent invalid data
- Commonly used values and constants, e.g. `host` , `port` for a `Database` class
 - Avoid repetition when creating objects

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class methods

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON



George Boorman
Curriculum Manager, DataCamp

Methods

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def give_raise(self, amount):
        self.salary += amount

# Unique attribute values
emp_one = Employee("John", 40000)
emp_one.give_raise(5000)
print(emp_one.salary)
```

45000

```
# Unique attribute values
emp_two = Employee("Jane", 60000)
emp_two.give_raise(5000)
print(emp_two.salary)
```

65000

Class methods

- Possible to define class methods
- Must have a narrow scope because they can't use object-level data

```
class MyClass:
    # Use a decorator to declare a class method
    @classmethod
    # cls argument refers to the class
    def my_awesome_method(cls, args...):
        # Do stuff here
        # Can't use any instance attributes
    # Call the class, not the object
    MyClass.my_awesome_method(args...)
```

- As with `self`, `cls` is a convention but any word will work

Alternative constructors

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def from_file(cls, filename):
        with open(filename, "r") as f:
            # Read the first line
            name = f.readline().strip()
            # Read the second line as integer
            salary = int(f.readline().strip())
        return cls(name, salary)
```

- Allow alternative constructors
- Can only have one `__init__()`
- Use class methods to create objects
- Use `return` to return an object
- `cls(...)` will call `__init__(...)`

Alternative constructors

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def from_file(cls, filename):
        with open(filename, "r") as f:
            name = f.readline().strip()
            salary = int(f.readline().strip())
        return cls(name, salary)
```

Employee.txt

```
1 John Smith
2 40000
```

```
# Create an employee without calling Employee()
emp = Employee.from_file("employee_data.txt")
print(emp.name)
```

```
John Smith
```

When to use class methods

- Alternative constructors
- Methods that don't require instance-level attributes
- Restricting to a single instance (object) of a class
 - Database connections
 - Configuration settings

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Class inheritance

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

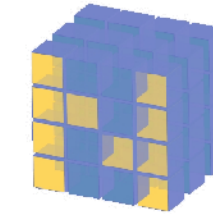


George Boorman
Curriculum Manager, DataCamp

Code reuse

1. Someone has already done it

- Packages are great for fixed functionality
- OOP is great for customizing functionality



NumPy



Code reuse

1. Someone has already done it

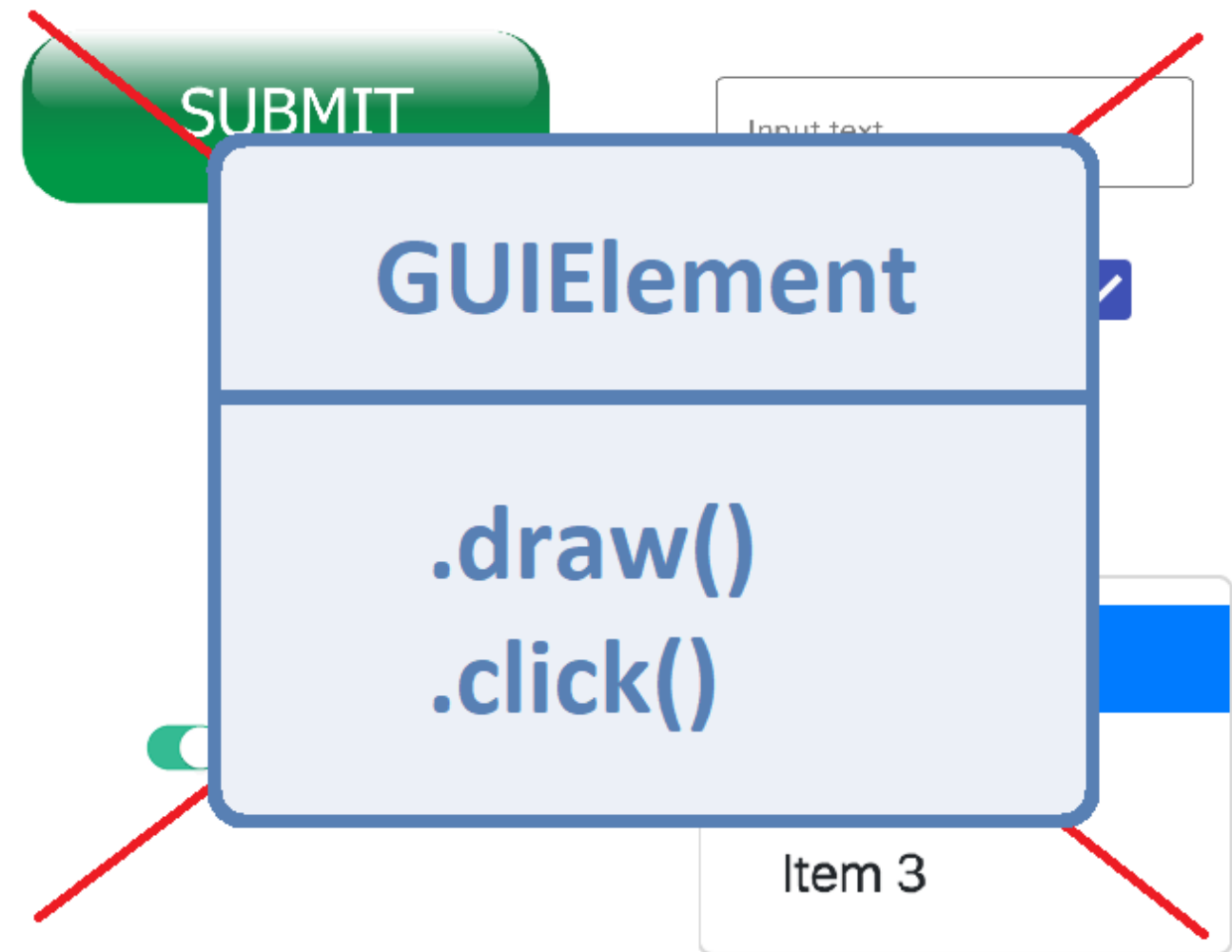
2. DRY: Don't Repeat Yourself

A collection of common web form elements:

- A green rounded rectangular button with the text "SUBMIT".
- Three radio buttons arranged vertically, labeled "One", "Two", and "Three". The first radio button is selected.
- A green toggle switch.
- A text input field with the placeholder text "Input text". Below it is the text "Helper text".
- A blue square checkbox with a white checkmark.
- A dark gray button with the text "Dropdown" and a downward arrow. Below it is a dropdown menu with three items: "Item 1" (highlighted in blue), "Item 2", and "Item 3".

Code reuse

1. Someone has already done it
2. DRY: Don't Repeat Yourself



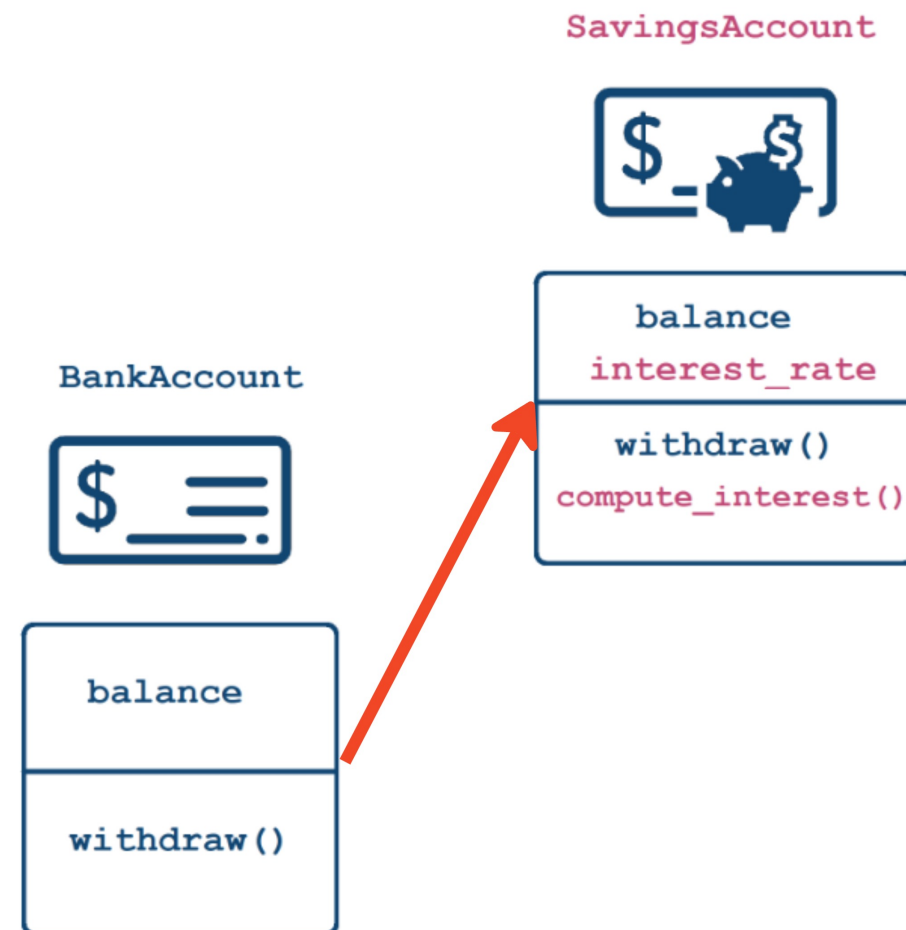
Inheritance

New class functionality = Old class functionality + extra

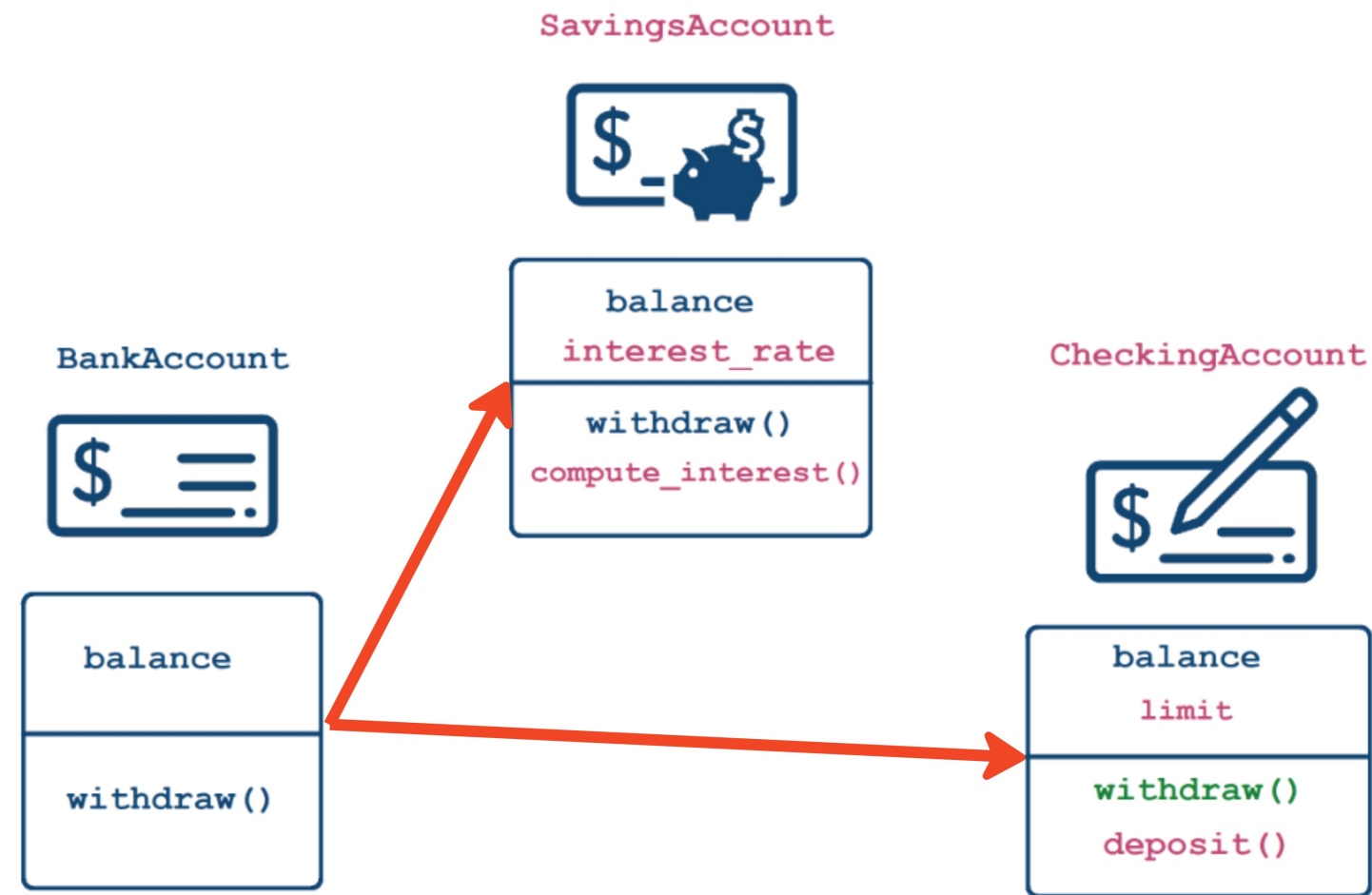
Example hierarchy



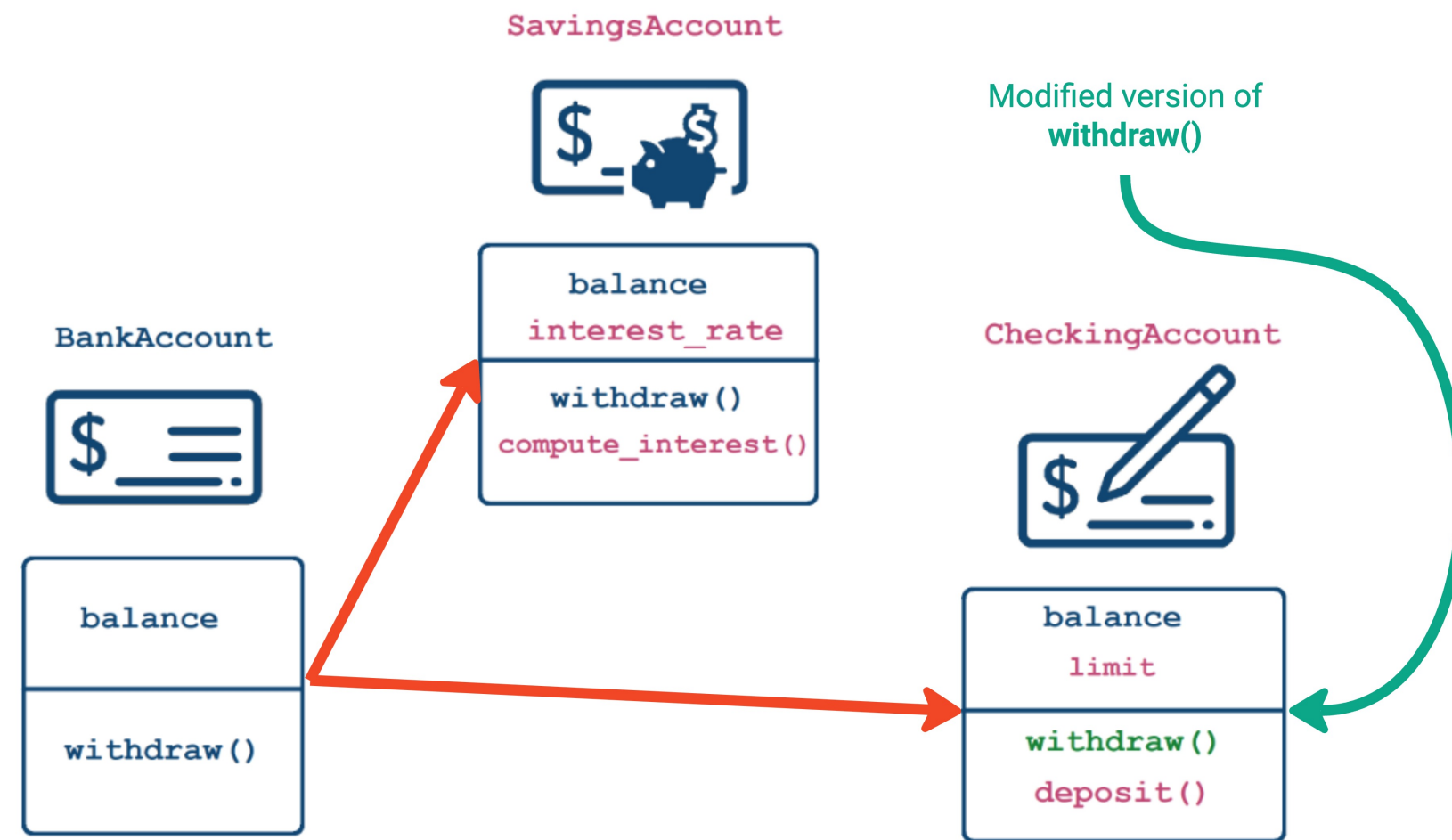
Example hierarchy



Example hierarchy



Example hierarchy



Implementing class inheritance

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance -= amount

# Class inheriting from BankAccount
class SavingsAccount(BankAccount):
    pass
```

- **BankAccount** : Parent class whose functionality is being extended/inherited
- **SavingsAccount** : Child/sub-class that will inherit the functionality and add more

Child class has all of the parent data

```
# Constructor inherited from BankAccount  
savings_acct = SavingsAccount(1000)  
type(savings_acct)
```

```
__main__.SavingsAccount
```

```
# Attribute inherited from BankAccount  
savings_acct.balance
```

```
1000
```

```
# Method inherited from BankAccount  
savings_acct.withdraw(300)
```

Inheritance: "is-a" relationship

A *SavingsAccount* is a *BankAccount*

(possibly with special features)

```
savings_acct = SavingsAccount(1000)
isinstance(savings_acct, SavingsAccount)
```

True

```
isinstance(savings_acct, BankAccount)
```

True

```
acct = BankAccount(500)
isinstance(acct, SavingsAccount)
```

False

```
isinstance(acct, BankAccount)
```

True

Let's practice!

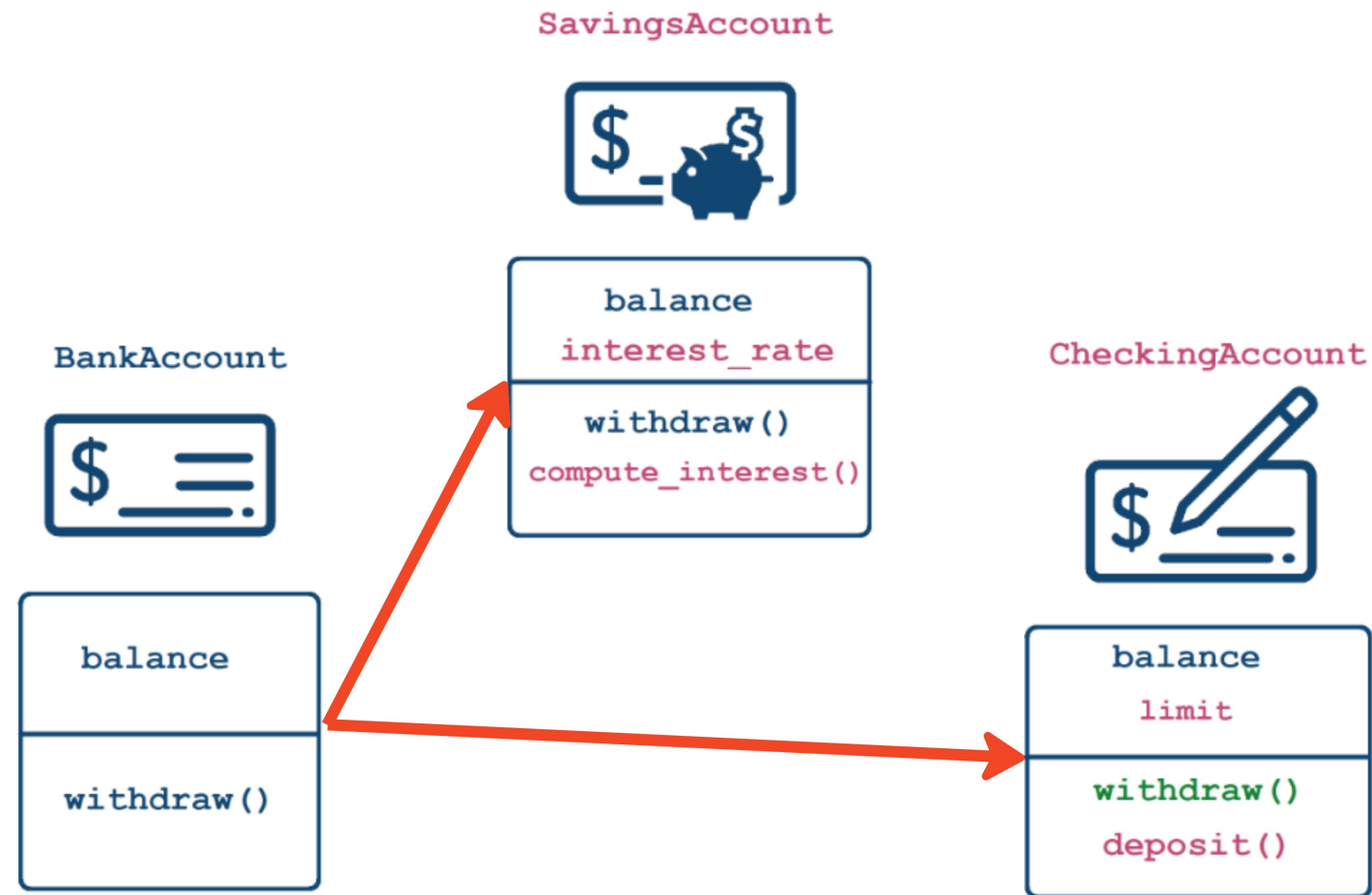
INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Customizing functionality via inheritance

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON



George Boorman
Curriculum Manager, DataCamp



What we have so far

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance -= amount

# Empty class inherited from BankAccount
class SavingsAccount(BankAccount):
    pass
```

Customizing constructors

```
class SavingsAccount(BankAccount):  
    # Constructor for SavingsAccount with an additional argument  
    def __init__(self, balance, interest_rate):  
        # Call the parent constructor using ClassName.__init__()  
        # self is a SavingsAccount but also a BankAccount  
        BankAccount.__init__(self, balance)  
        # Add more functionality  
        self.interest_rate = interest_rate
```

- Can run constructor of the parent class first by `Parent.__init__(self, args...)`
- Add more functionality
- Don't *have* to call the parent constructor

Create objects with a customized constructor

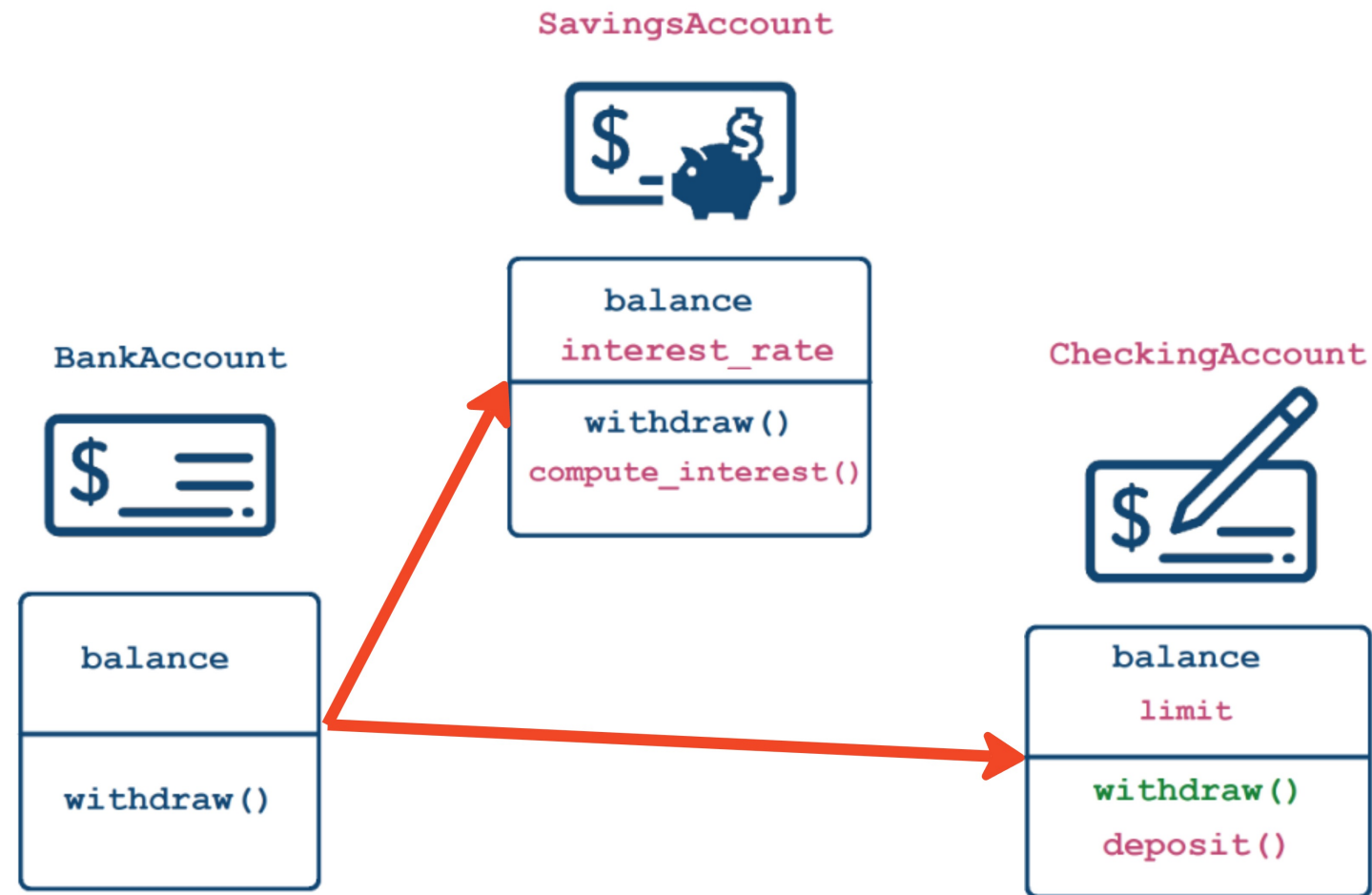
```
# Construct the object using the new constructor  
acct = SavingsAccount(1000, 0.03)  
acct.interest_rate
```

```
0.03
```

Adding functionality

- Add methods as usual
- Can use the data from both the parent and the child class

```
class SavingsAccount(BankAccount):  
    def __init__(self, balance, interest_rate):  
        BankAccount.__init__(self, balance)  
        self.interest_rate = interest_rate  
  
    # New functionality  
    def compute_interest(self, n_periods=1):  
        return self.balance * ( (1 + self.interest_rate) ** n_periods - 1)
```



Adding a second child class

```
class CheckingAccount(BankAccount):
    def __init__(self, balance, limit):
        BankAccount.__init__(self, balance) # Call the ParentClass constructor
        self.limit = limit
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount, fee=0): # New fee argument
        if amount <= self.limit:
            BankAccount.withdraw(self, amount + fee)
        else:
            pass # Won't run if the condition isn't met
```

```
check_acct = CheckingAccount(1000, 25)
```

```
# Will call withdraw from CheckingAccount  
check_acct.withdraw(200)
```

```
# Will call withdraw from CheckingAccount  
check_acct.withdraw(200, fee=15)
```

```
bank_acct = BankAccount(1000)
```

```
# Will call withdraw from BankAccount  
bank_acct.withdraw(200)
```

```
# Will produce an error  
bank_acct.withdraw(200, fee=15)
```

```
TypeError: withdraw() got an unexpected  
keyword argument 'fee'
```

- Violates polymorphism
 - Parent / child classes have different methods

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON