Ruby 3

OPL 2/66

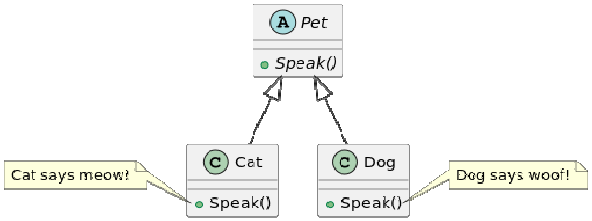
Outline

- Dynamic dispatch
- Method overriding
- Exercise

Dynamic dispatch

- **dynamic dispatch** is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.
- Dynamic dispatch contrasts with *static dispatch*, in which the implementation of a polymorphic operation is selected at compile time.
- The purpose of dynamic dispatch is to defer the selection of an appropriate implementation until the run time type of a parameter (or multiple parameters) is known.

Example




```ruby
class Pet
    def speak
    end
end

class Dog < Pet
    def speak
        puts "Woof"
    end
end

class Cat < Pet
    def speak
        puts "Meow"
    end
end

def speak(pet)
    pet.speak
end

dog = Dog.new
cat = Cat.new

speak(dog)
speak(cat)
```

Ruby method lookup

The semantics for method calls also known as message sends

e0.m(e1,…,en)

- Evaluate e0, e1, …, en to objects obj0, obj1, …, objn
  - As usual, may involve looking up self, variables, fields, etc.
- Let C be the class of obj0 (every object has a class)
- If m is defined in C, pick that method, else recur with the superclass of C unless C is already Object
  - If no m is found, call method_missing instead
    - Definition of method_missing in Object raises an error
- Evaluate body of method picked:
  - With formal arguments bound to obj1, …, objn
  - With self bound to obj0 -- this implements dynamic dispatch!

A simple example, part 2

- In Ruby (and other OOP languages), subclasses can change the behavior of methods they do not override

```
class A
  def even x
    if x==0 then true  else odd  (x-1) end
  end
  def odd x
    if x==0 then false else even (x-1) end
  end
end
class B < A  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses even if it is not overridden

- Maybe on purpose, maybe by mistake
- Observable behavior includes calls-to-overridable methods

- So *harder* to reason about "the code you're looking at"
  - Can avoid by disallowing overriding
    - "private" or "final" methods

- So *easier* for subclasses to affect behavior without copying code
  - Provided method in superclass is not modified later

Exercise: Banking System using Object-Oriented Programming in Ruby

1. Create a BankAccount class:
   - It should have attributes such as account_number, balance, owner_name, etc.
   - Implement methods like deposit, withdraw, and display_balance.
   - Ensure validation in the withdraw method to avoid overdrawing.
2. Create a Transaction class:
   - This class should represent individual transactions made by customers.
   - Each transaction should have attributes like amount, date, type (deposit/withdrawal), etc.
   - Implement a method to display transaction details.
3. Use inheritance:
   - Create different types of bank accounts by inheriting from BankAccount (e.g., SavingsAccount, CheckingAccount).
   - Add specific functionalities to these subclasses.
4. Create instances and perform transactions:
   - Instantiate objects of the classes created above and demonstrate how deposits, withdrawals, and account details work.
   - Show transactions and their details.

Example usage

```ruby
# Example Usage
account1 = BankAccount.new("123456", "Alice")
account1.deposit(1000)
account1.withdraw(500)
account1.display_balance

transaction1 = Transaction.new(1000, "Deposit")
transaction1.display_transaction
```

Reference

- Dan Grossman's PL lecture 21