# Software Reuse with Inheritance

**Maurice Naftalin**
Java Champion, JavaOne Rock Star
Author: *Mastering Lambdas, Java Generics and Collections*
@mauricenaftalin

# Module Overview

Why Inheritance?

Encapsulating Variation by Subclassing

The Liskov Substitution Principle

Overriding and Dynamic Dispatch

Accessing Overridden Methods

Understanding Inheritance

Abstract Classes

The Open-Closed Principle

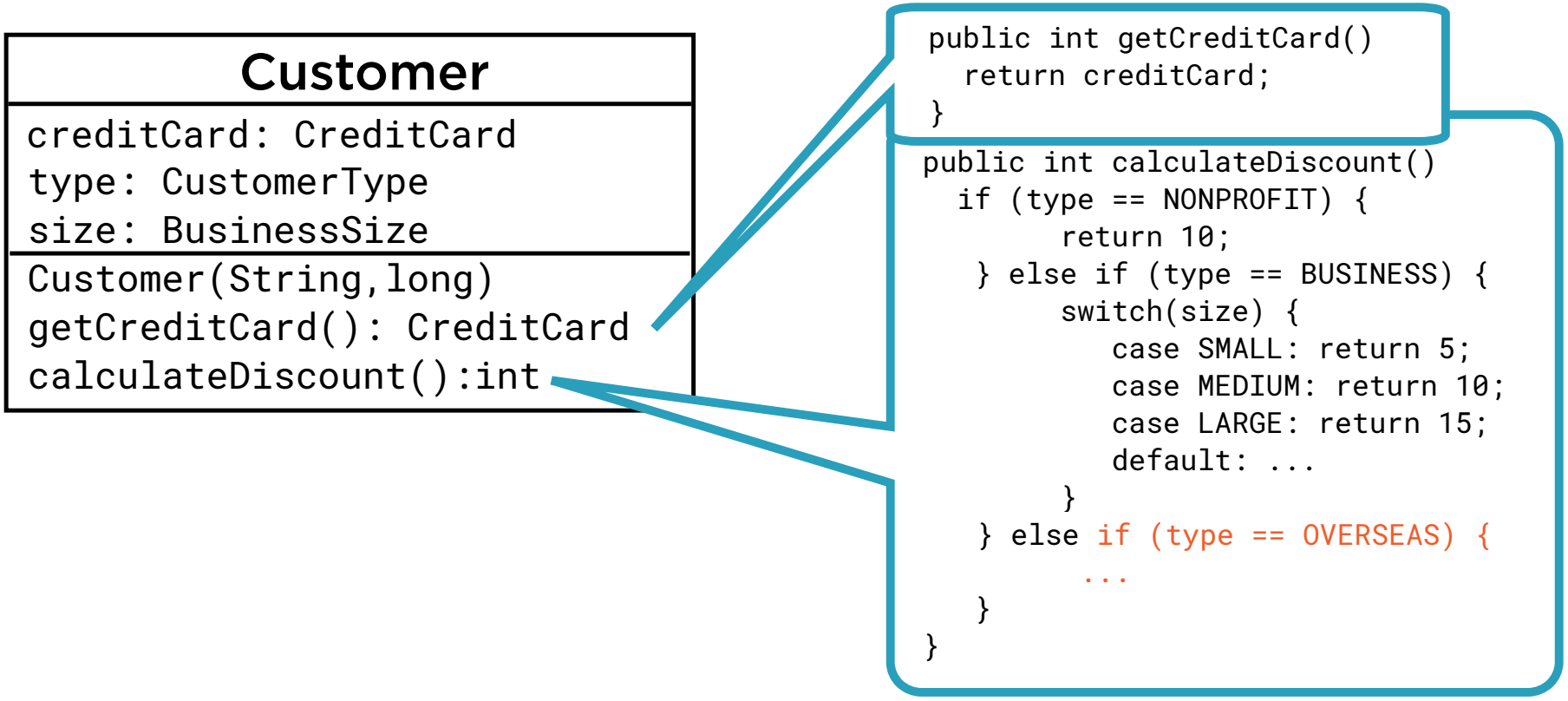

Vertebrates (animals with backbone)

Mammals Fish Birds

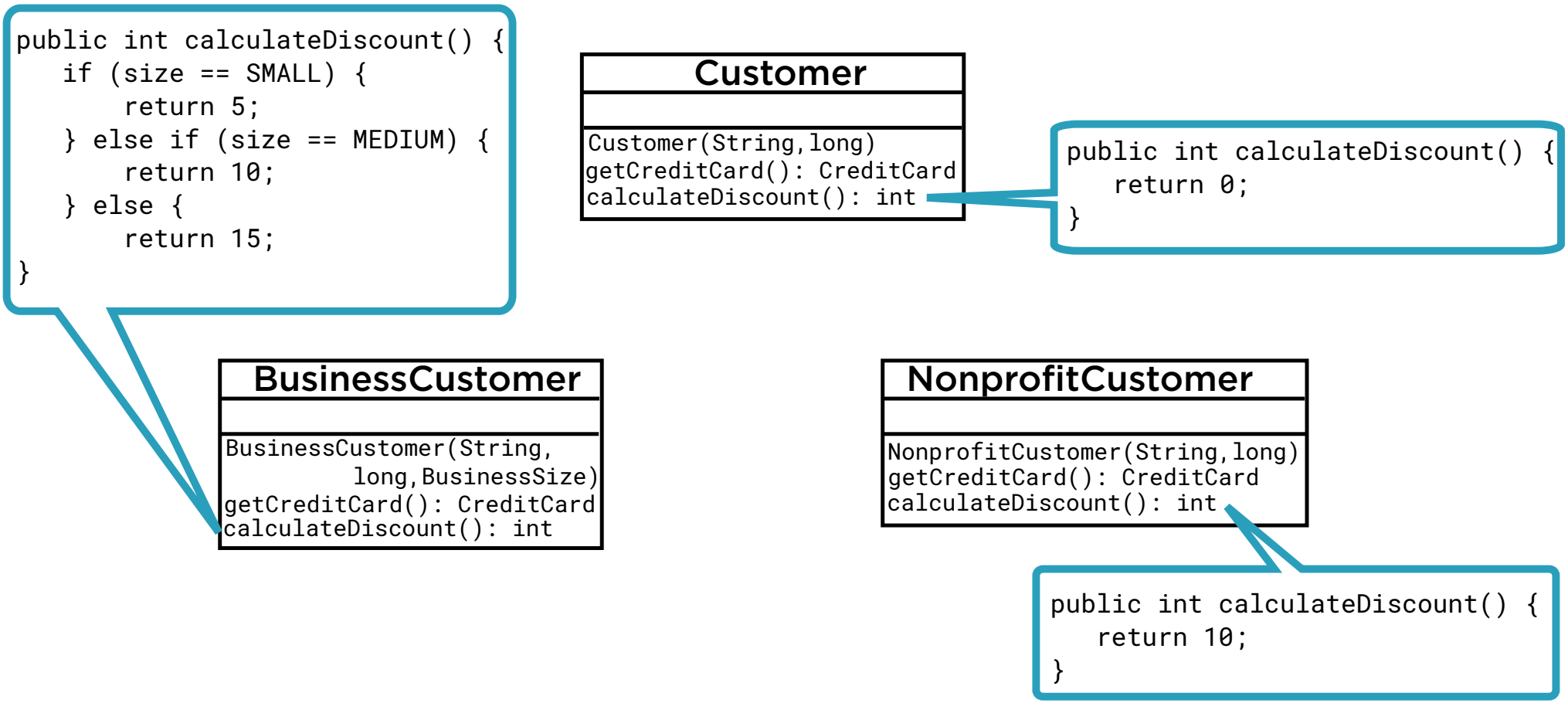
Dogs (members of species *Canis familiaris*)

Salukis

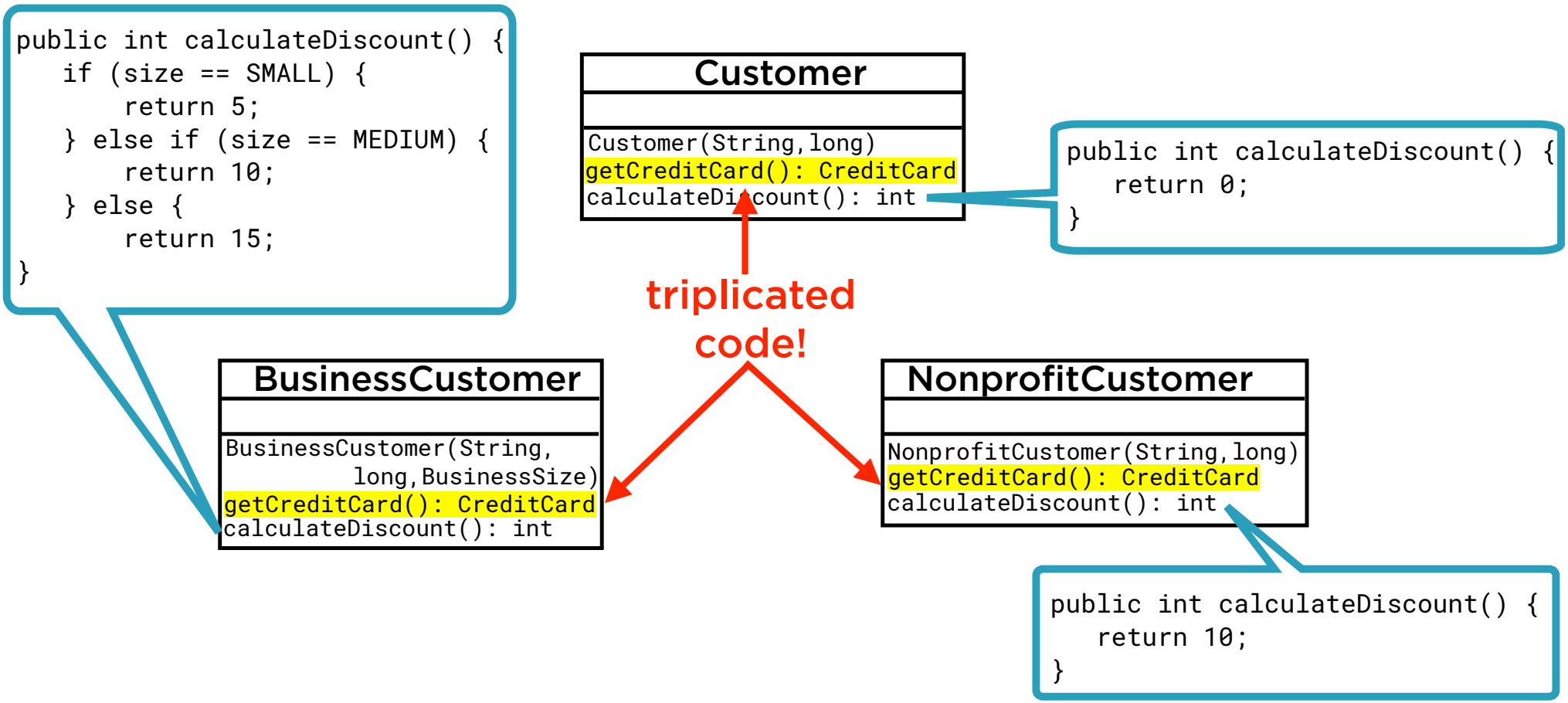
# New Responsibilities — the Old-fashioned Way



# New Responsibilities — with Copy-and-paste

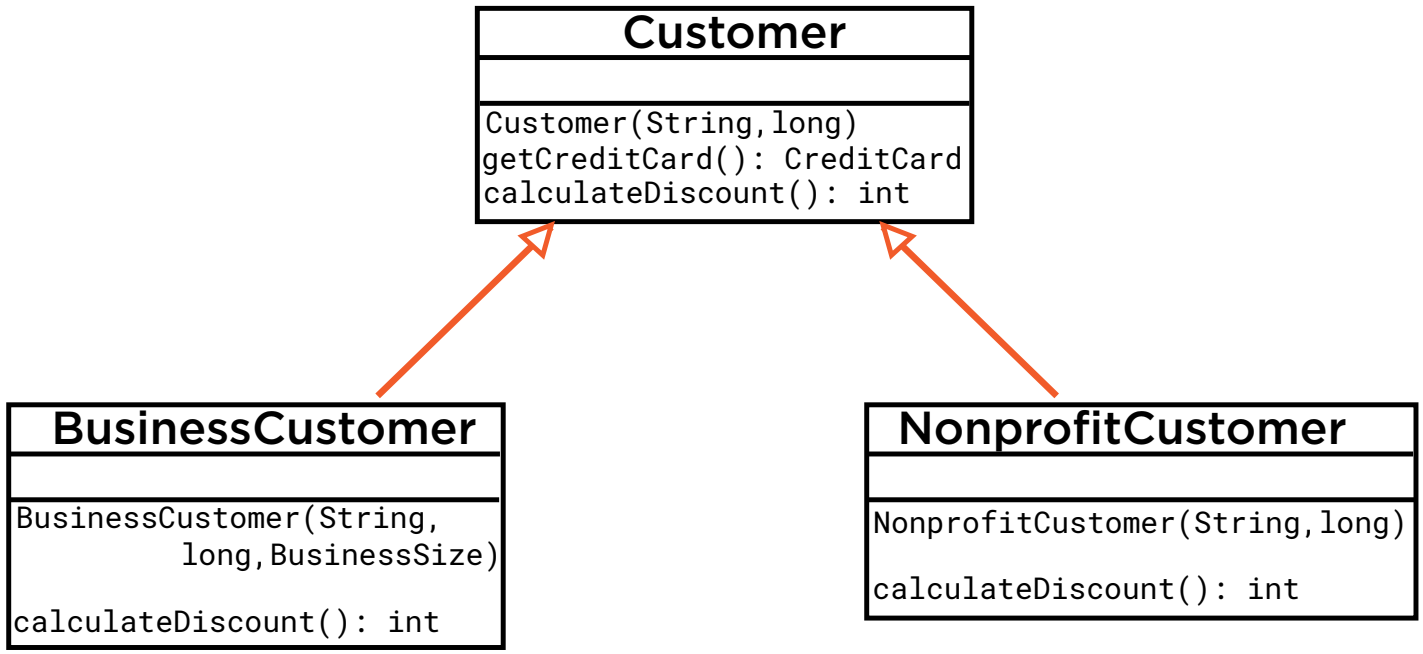


# New Responsibilities — with Copy-and-paste

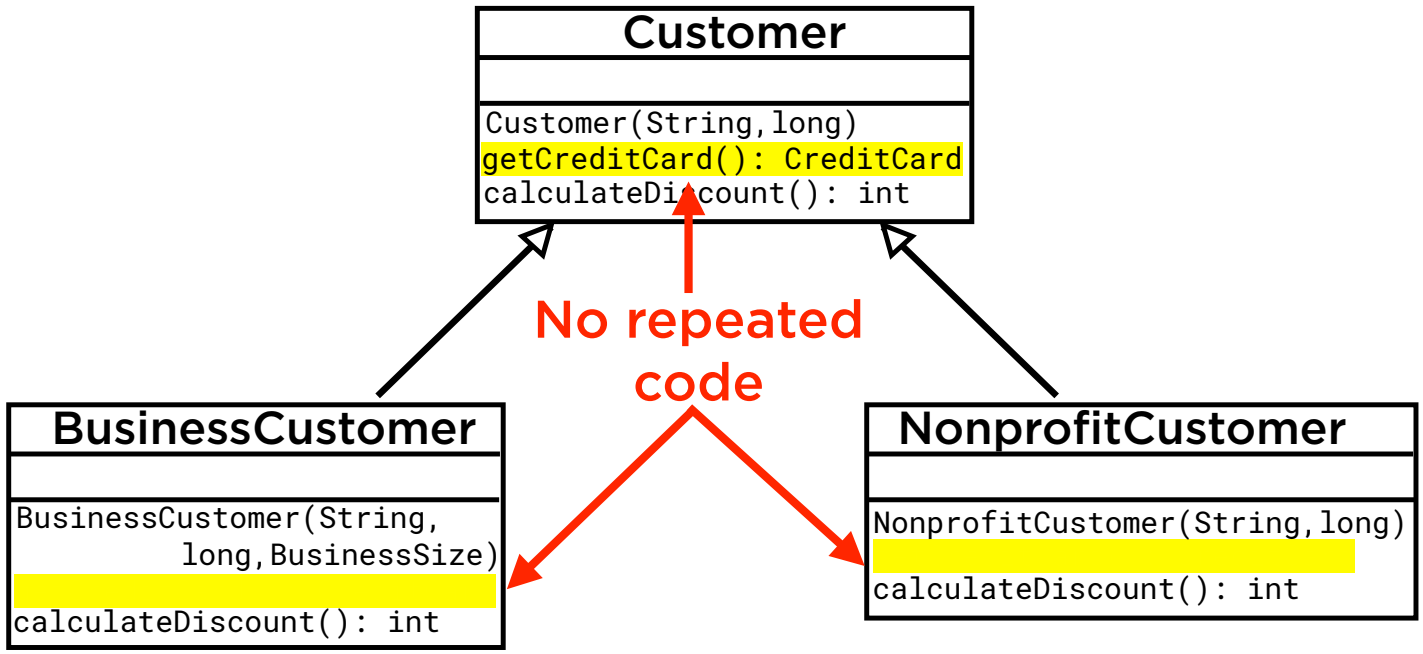


# DRY:
# Don’t Repeat Yourself!

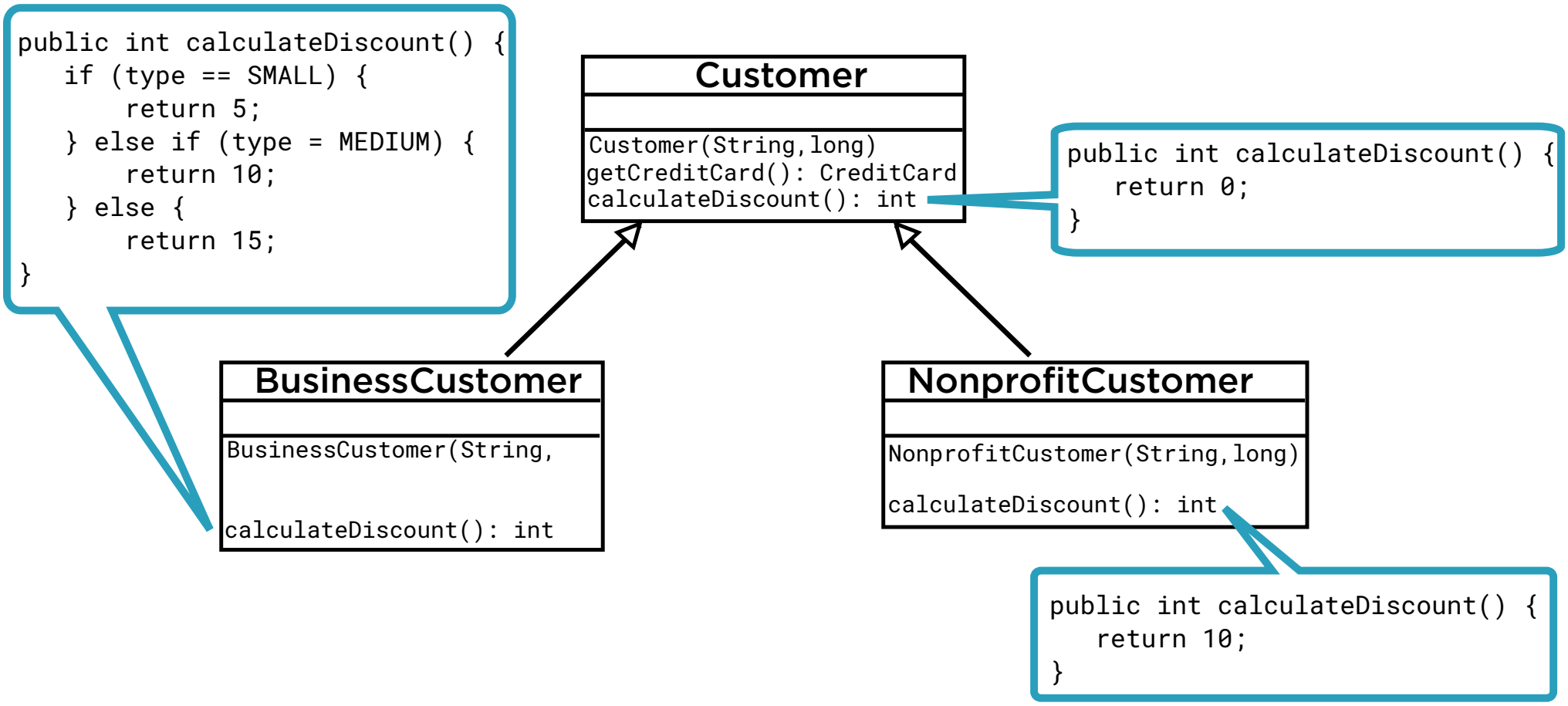
# New Responsibilities — with Inheritance



# New Responsibilities — with Inheritance



# New Responsibilities — with Inheritance



```
Customer cust =
        new Customer("John Doe", 5420793615183044L);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```

# Encapsulation of Subclass Variation

**The variable** `cust` **can refer to an instance of** `Customer`

```
Customer cust =
        new Customer(“John Doe", 5420793615183044L);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```

# Encapsulation of Subclass Variation

**The variable** `cust` **can refer to an instance of** `Customer`

```
Customer cust =
        new BusinessCustomer(“Acme Products", 5420793615183044L, SMALL);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```

# Encapsulation of Subclass Variation

**The variable** `cust` **can refer to an instance of** `Customer`

- **or an instance of a** `Customer` **subclass**

```
Customer cust =
        new BusinessCustomer(“Acme Products", 5420793615183044L, SMALL);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```

# Encapsulation of Subclass Variation

**The variable** `cust` **can refer to an instance of** `Customer`

- **or an instance of a** `Customer` **subclass**

Liskov Substitution Principle:

Wherever a superclass object is expected, you can always use a subclass object instead

```
Customer cust =
        new BusinessCustomer(“Acme Products", 5420793615183044L, SMALL);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```

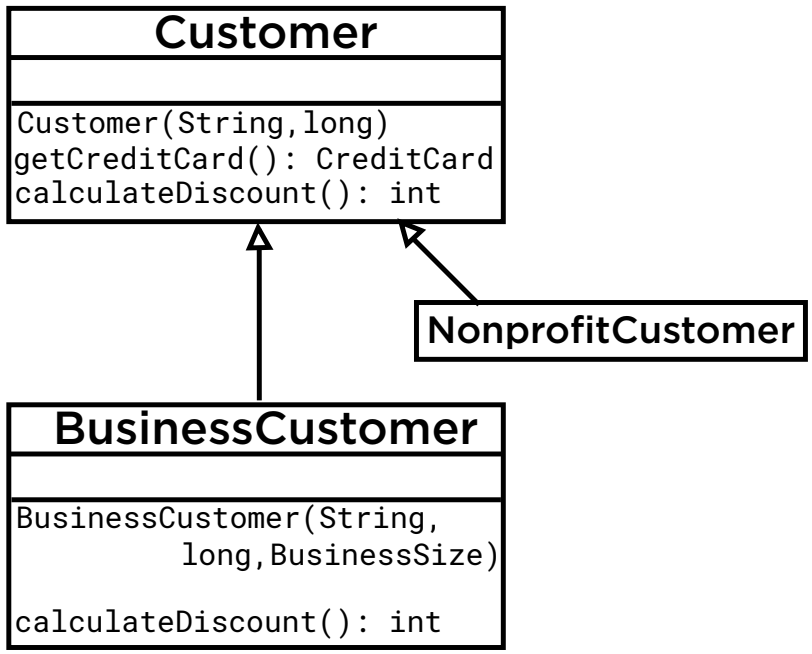
# Encapsulation of Subclass Variation

**The variable** `cust` **can refer to an instance of** `Customer`

- **or an instance of a** `Customer` **subclass**

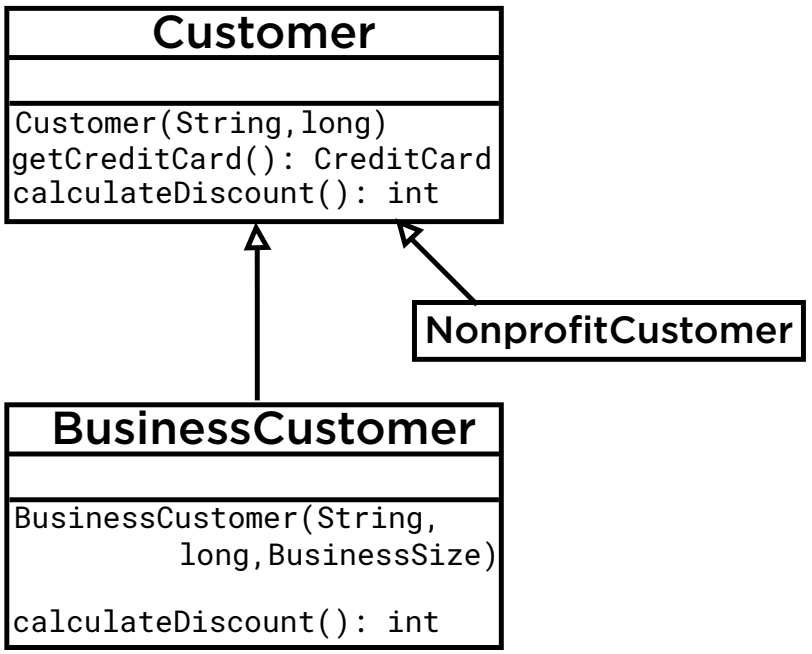
# How Overriding Works – Compile Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



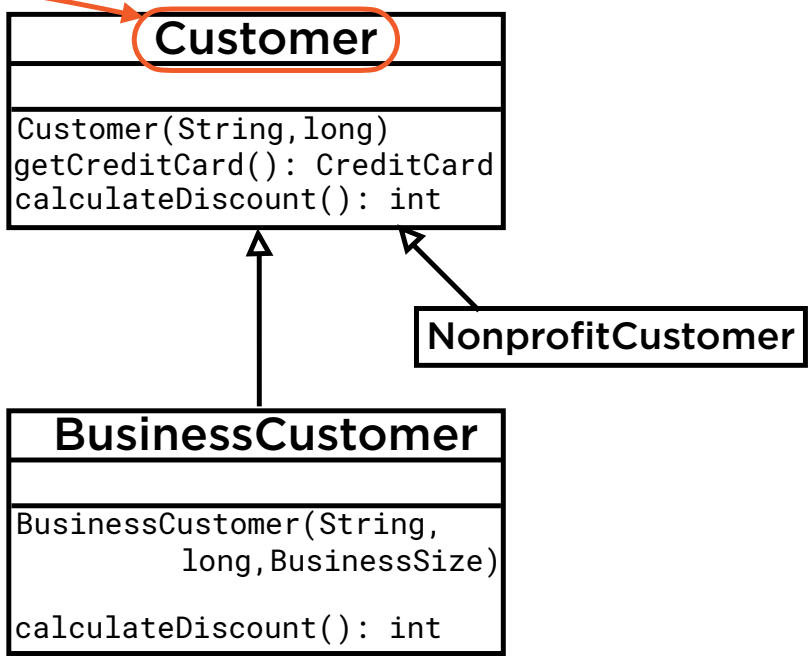
# How Overriding Works – Compile Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



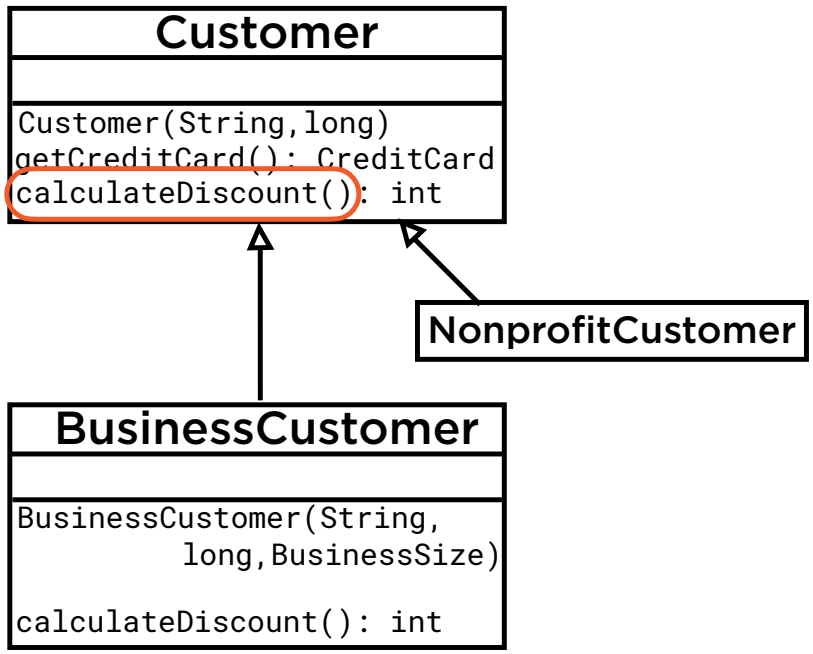
# How Overriding Works – Compile Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



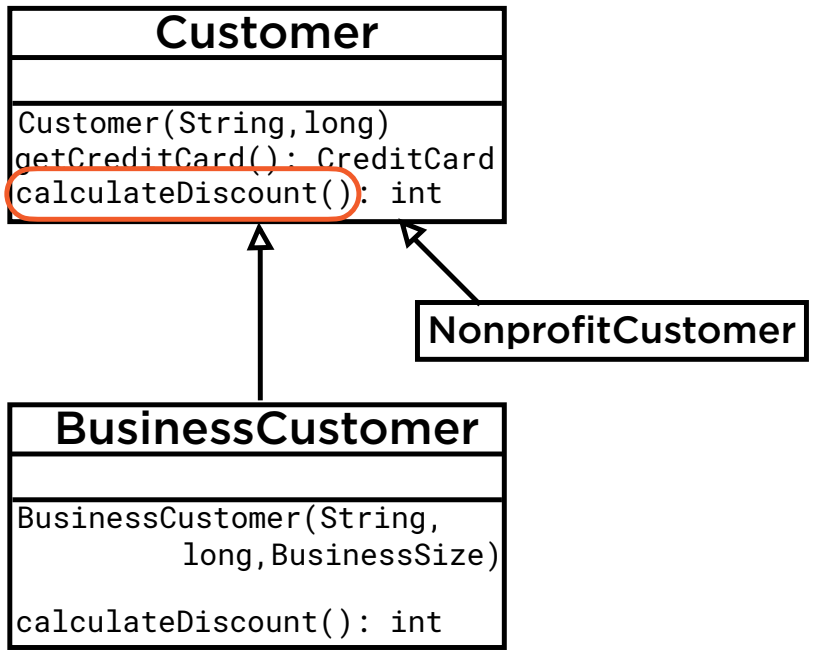
# How Overriding Works – Compile Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



# How Overriding Works – Compile Time

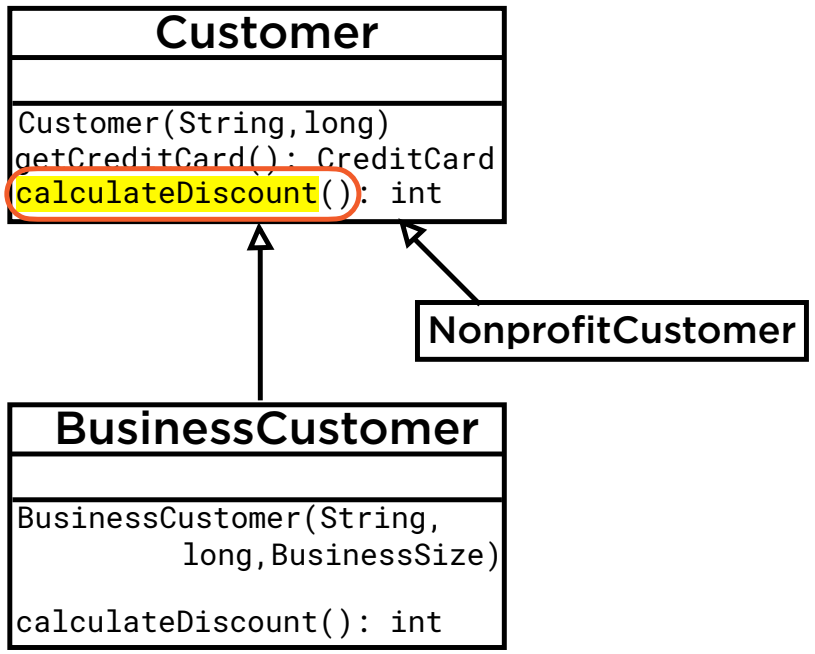
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

# How Overriding Works – Compile Time

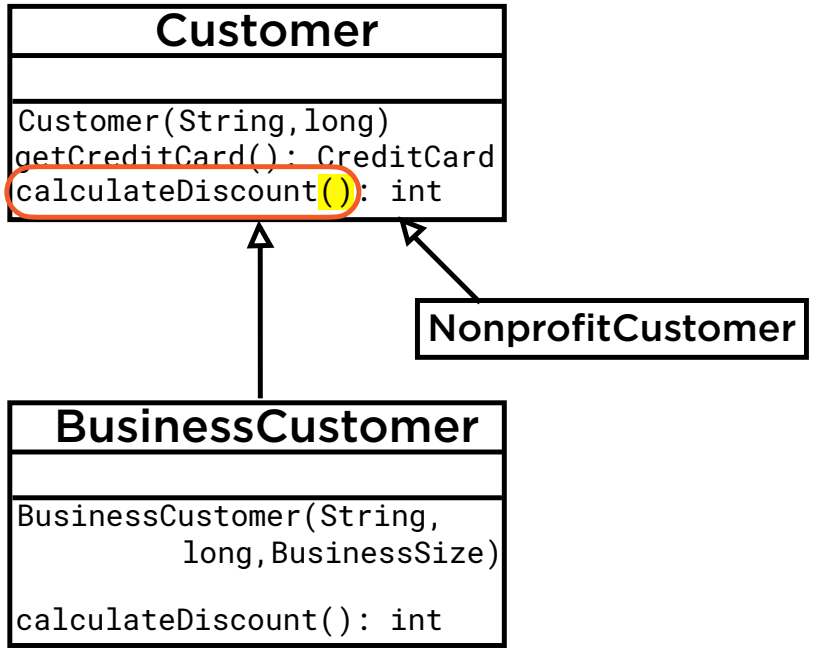
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

# How Overriding Works – Compile Time

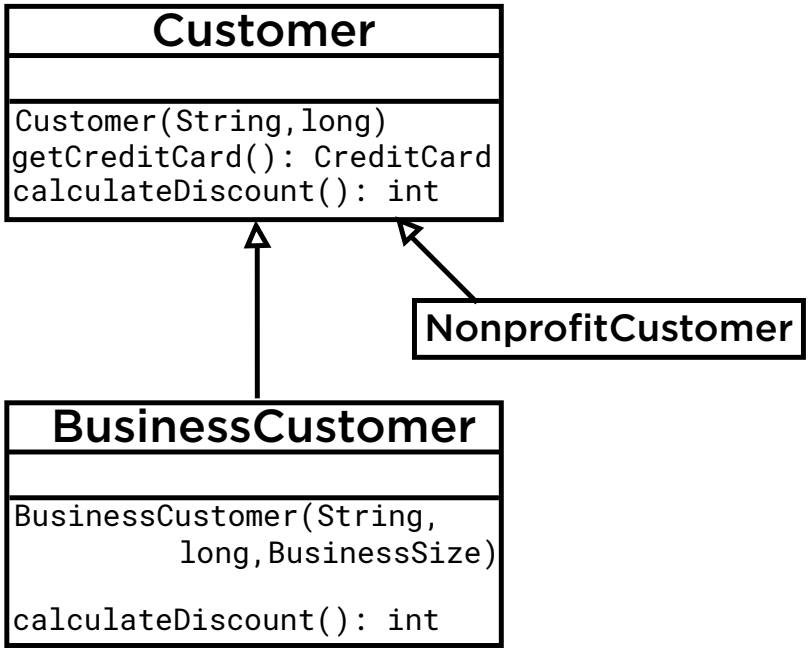
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

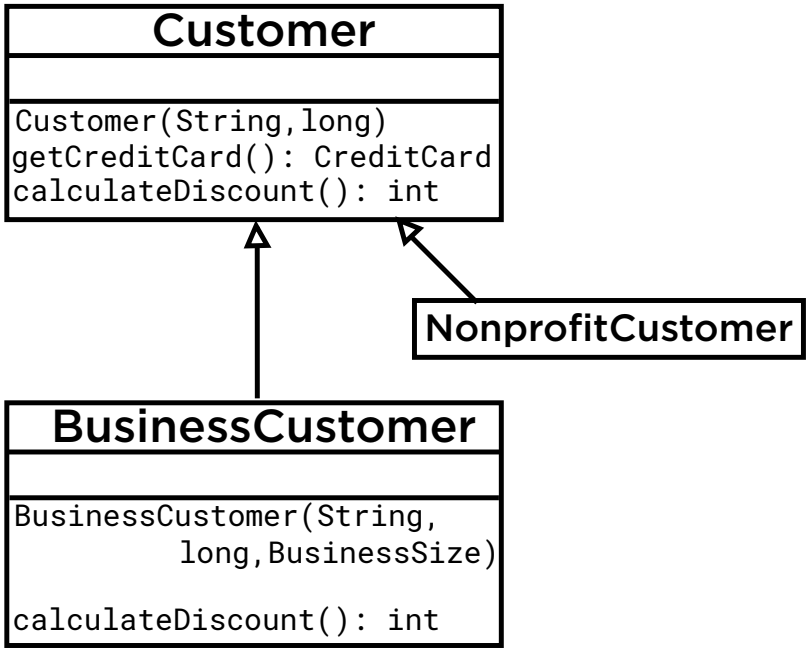
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



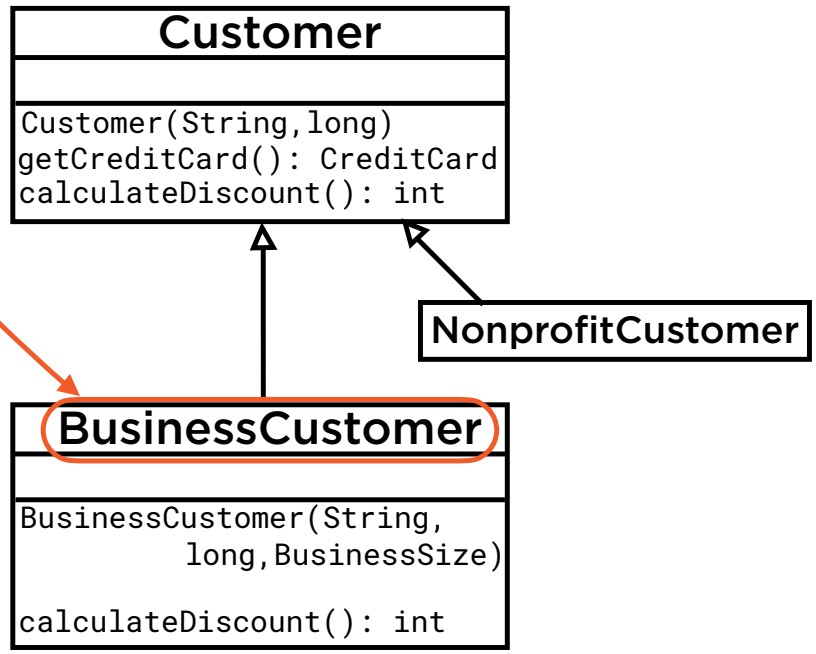
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



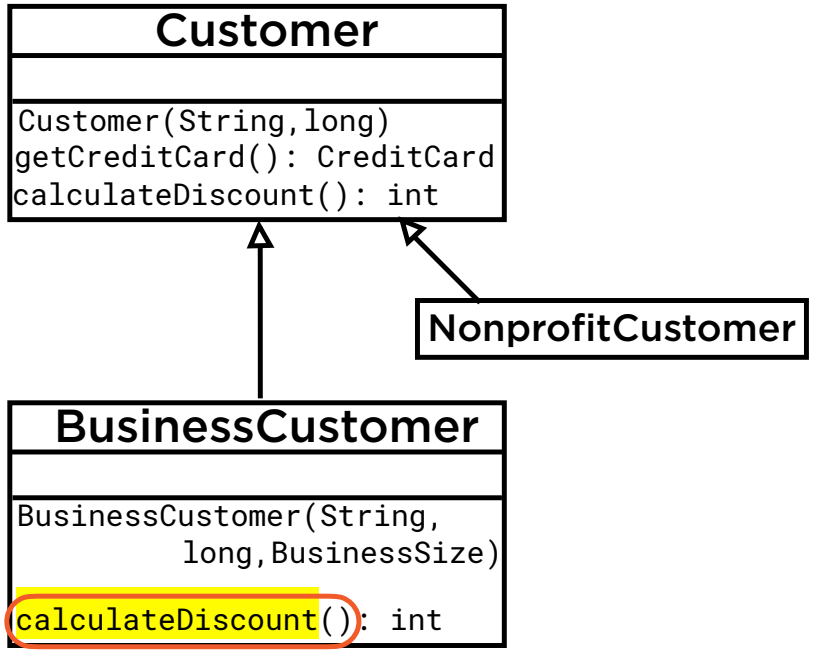
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



# How Overriding Works – Compile Time

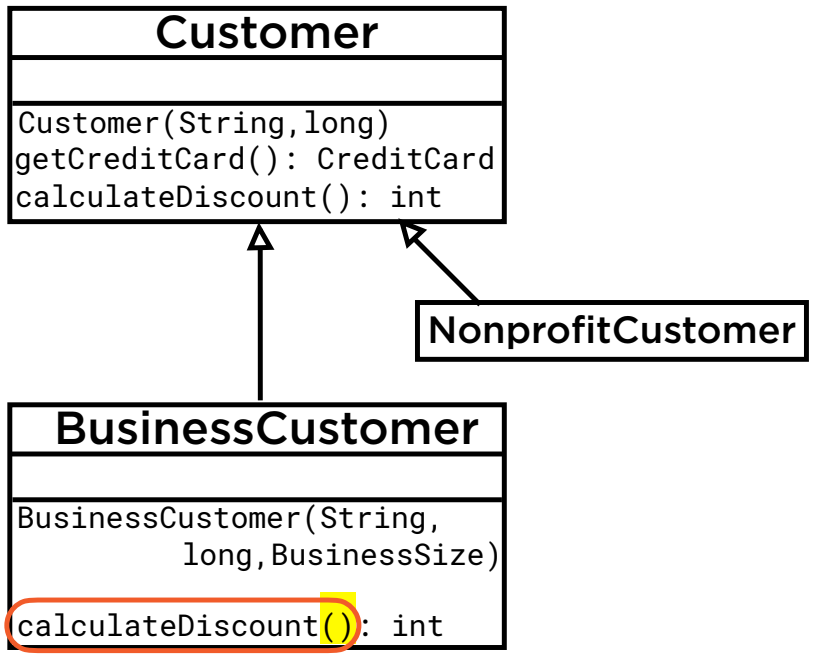
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

# How Overriding Works – Compile Time

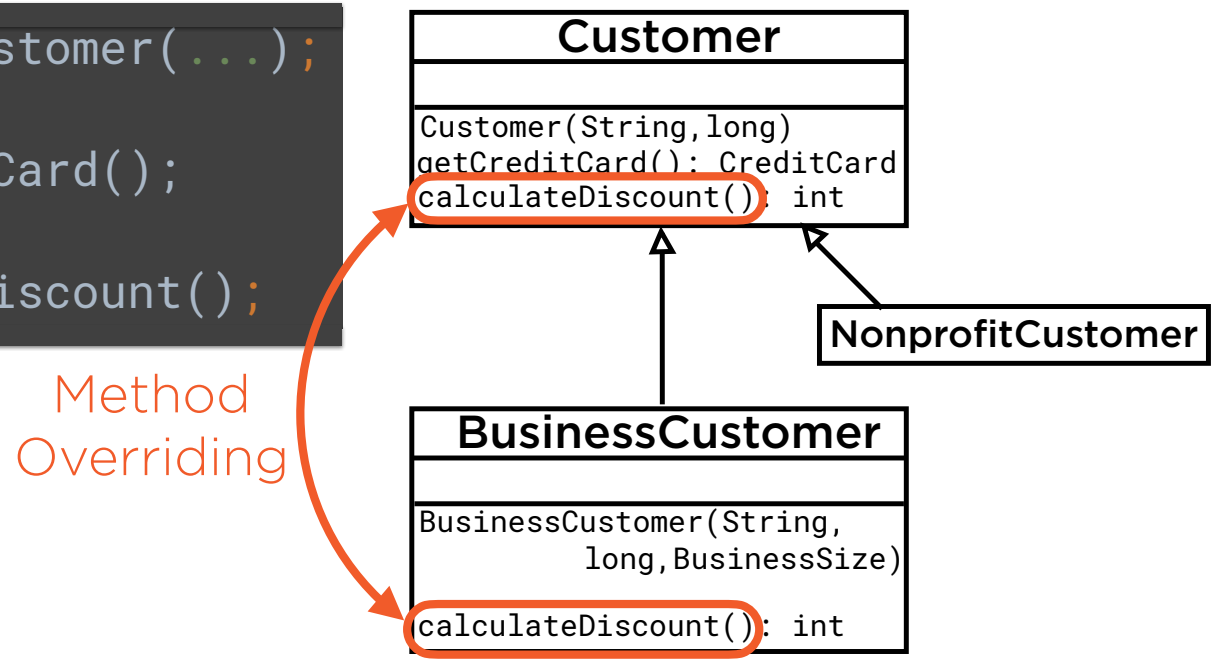
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

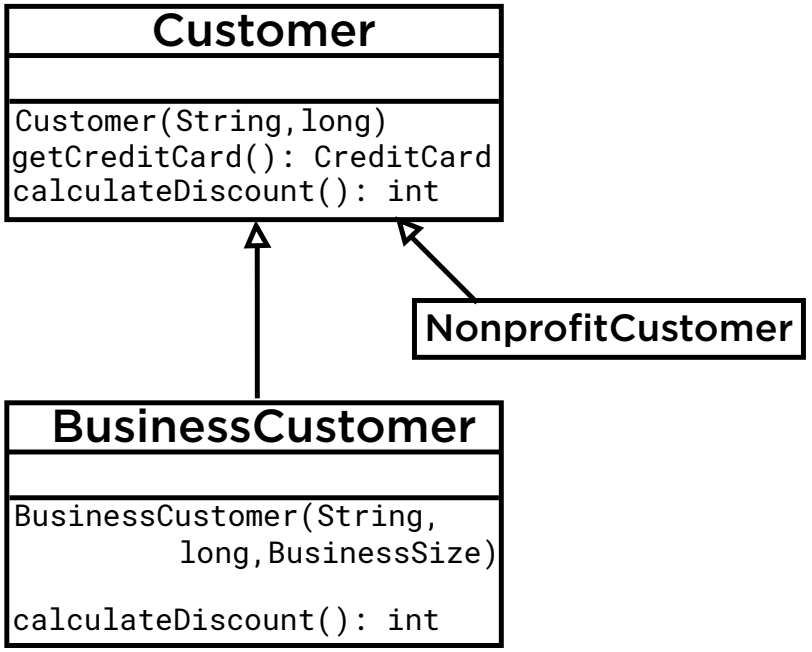
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



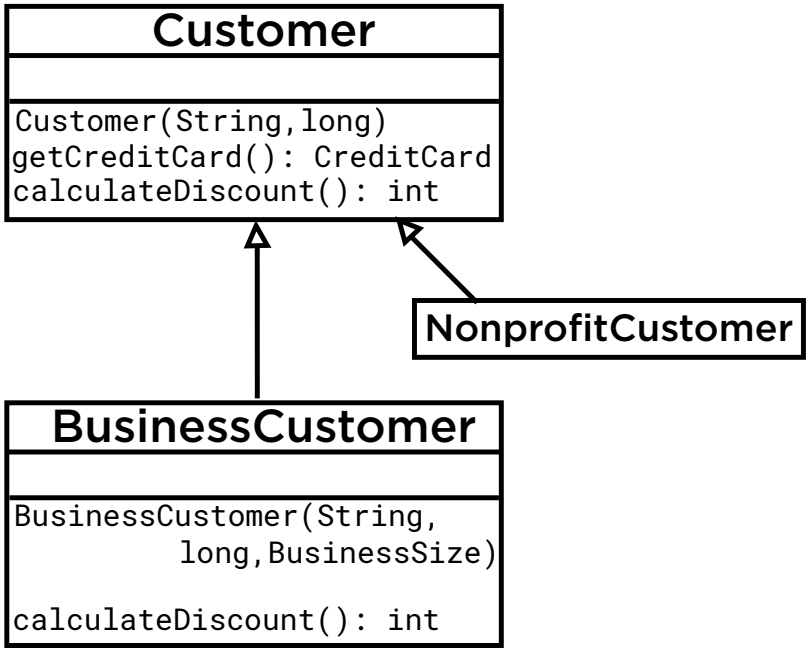
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



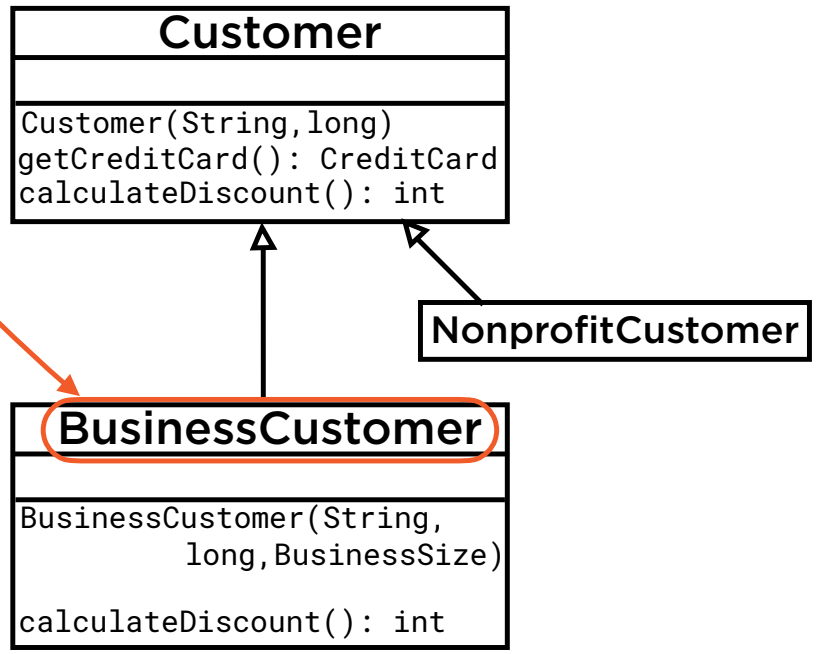
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



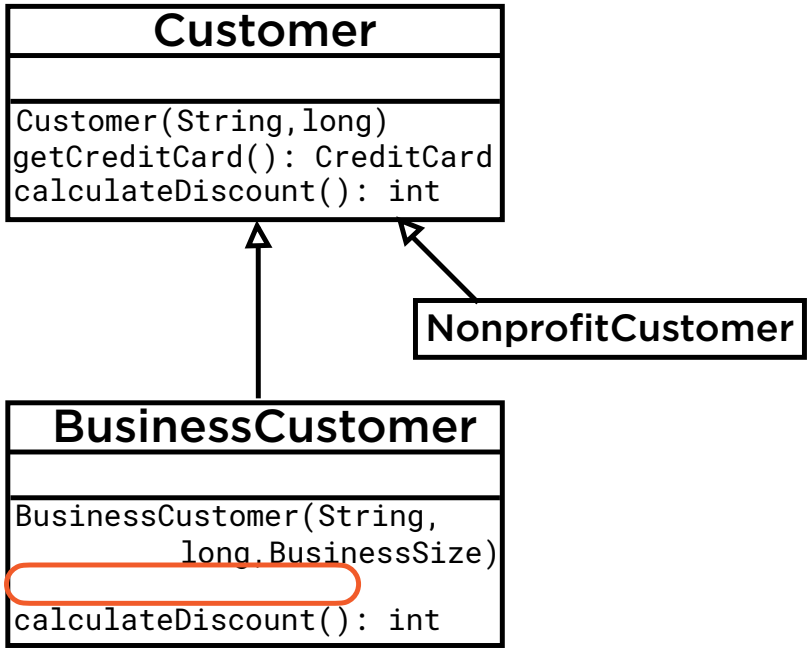
# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```



# How Overriding Works – Run Time

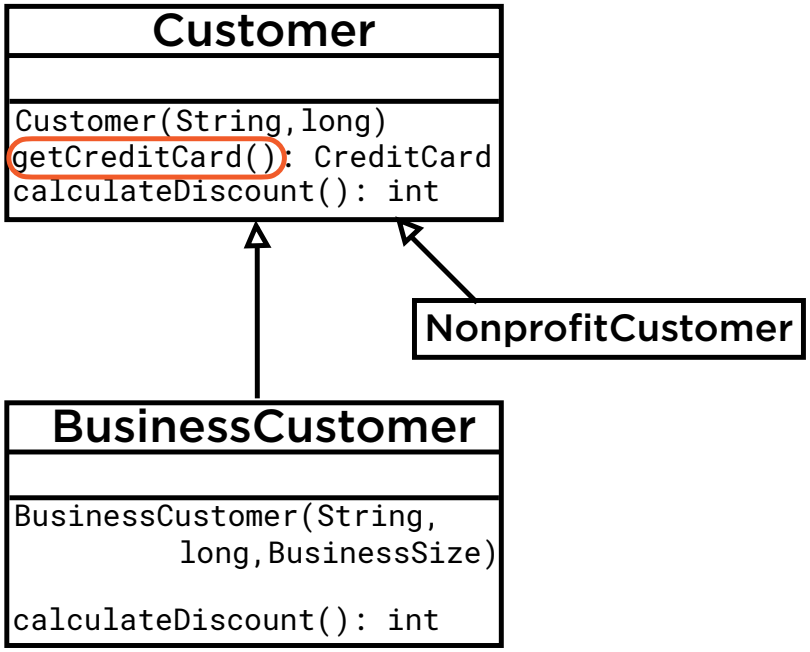
```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

# How Overriding Works – Run Time

```
Customer cust = new BusinessCustomer(...);

CreditCard cc = cust.getCreditCard();

int discount = cust.calculateDiscount();
```




Signature = method name + parameter list

# Dynamic Dispatch

is the method of selecting at run time which implementation of a method should be called.

multiply-defined
(polymorphic)

# Demo: Dynamic Dispatch

**Using dynamic dispatch to select the right overriding method**

# Dynamic Dispatch

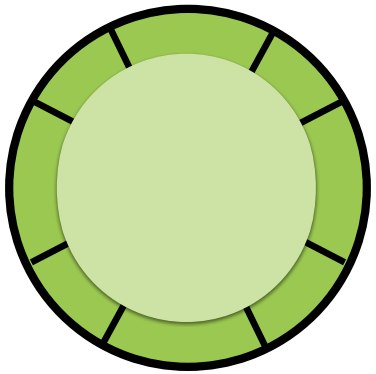
jshell

```
jshell> class Customer {
   ...>   public String toString() {
   ...>     return "Customer";
   ...>   }
   ...> }
|  created class Customer

jshell> class BusinessCustomer extends Customer {
   ...>   public String toString() {
   ...>     return "BusinessCustomer";
   ...>   }
   ...> }
|  created class BusinessCustomer

jshell> Customer cust = new Customer()
cust ==> Customer

jshell> System.out.print(cust)
Customer
jshell> cust = new BusinessCustomer()
cust ==> BusinessCustomer

jshell> System.out.print(cust)
BusinessCustomer
jshell>
```

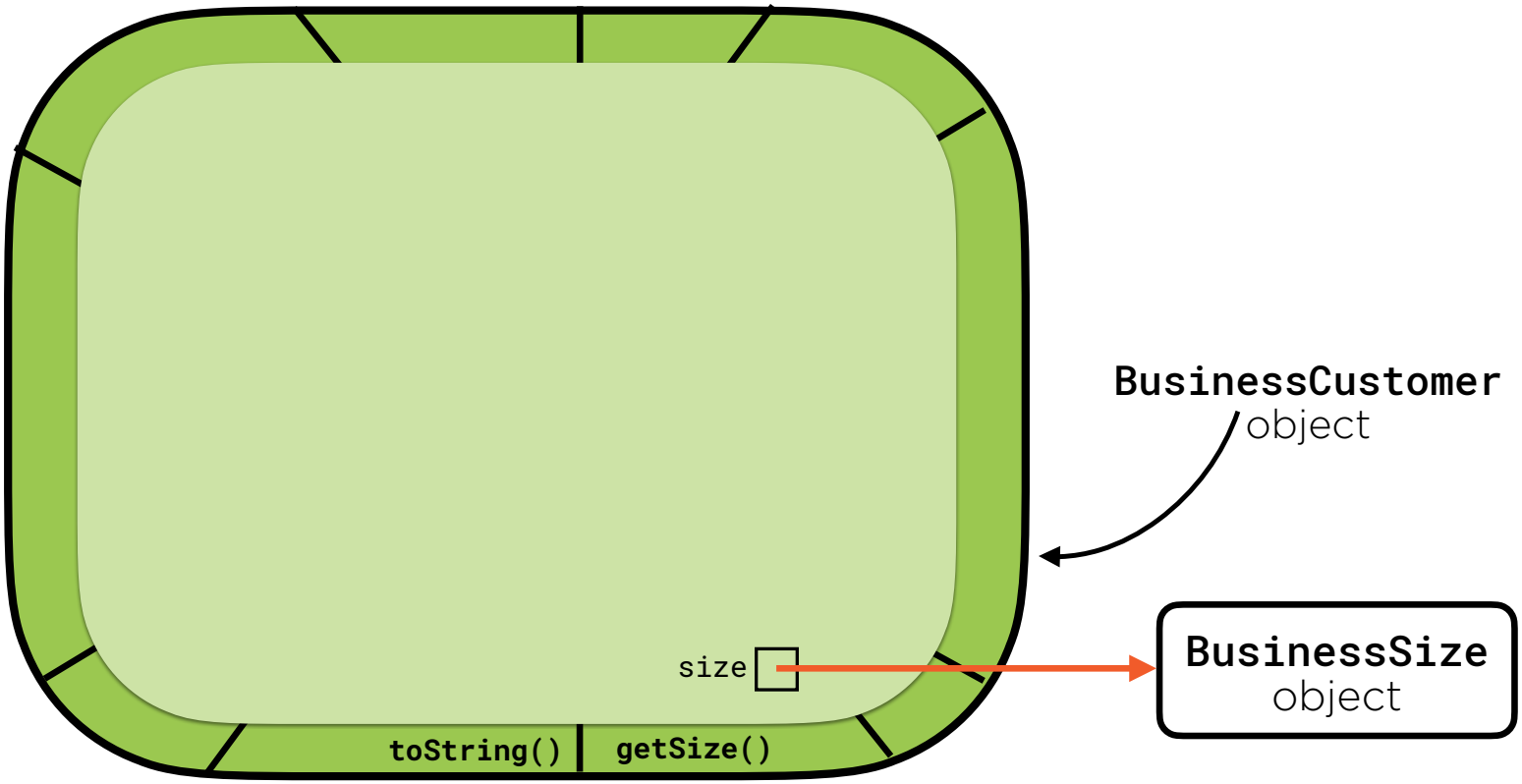
# Demo: Using a Superclass Method

**Accessing overridden methods**

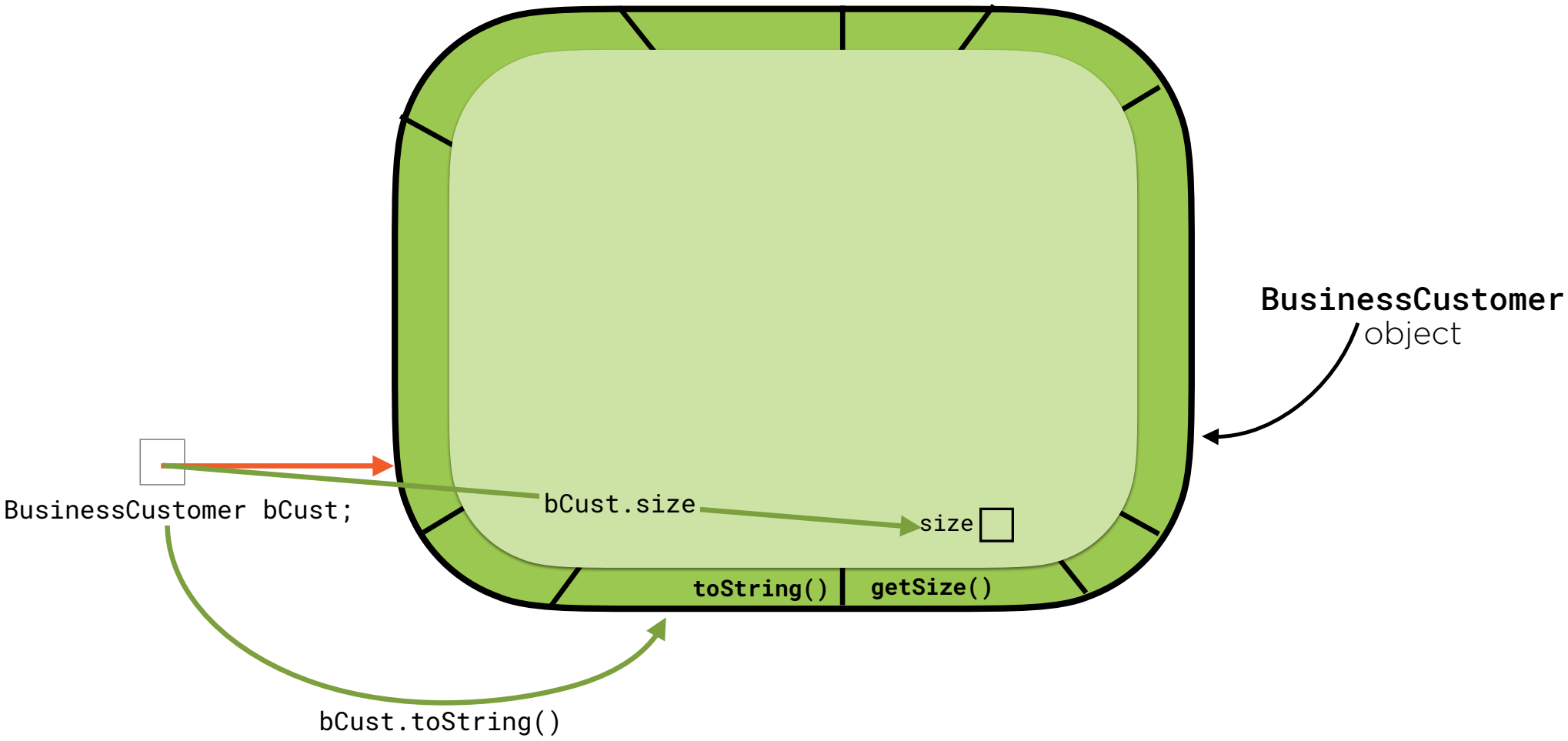
# Understanding Inheritance



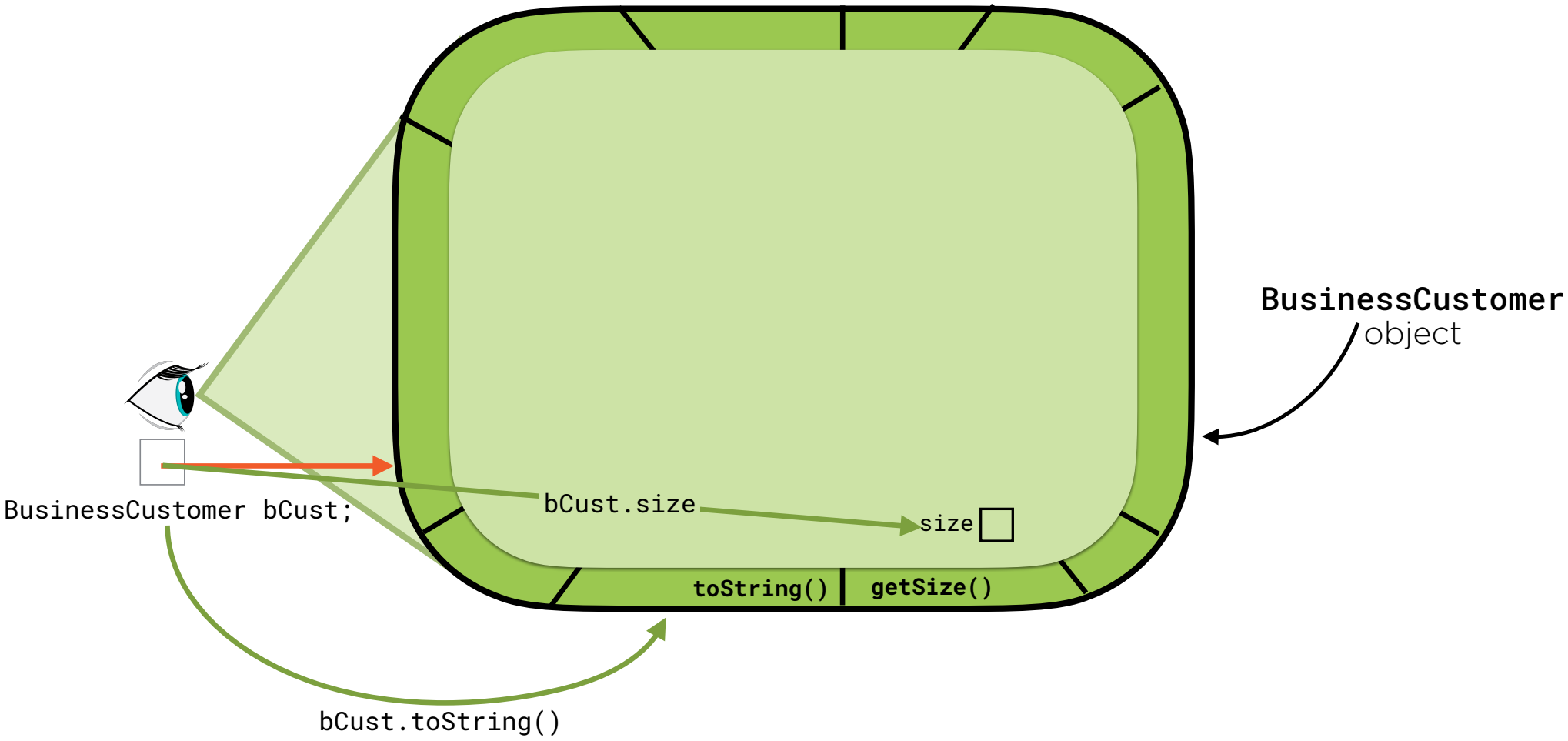
# Understanding Inheritance



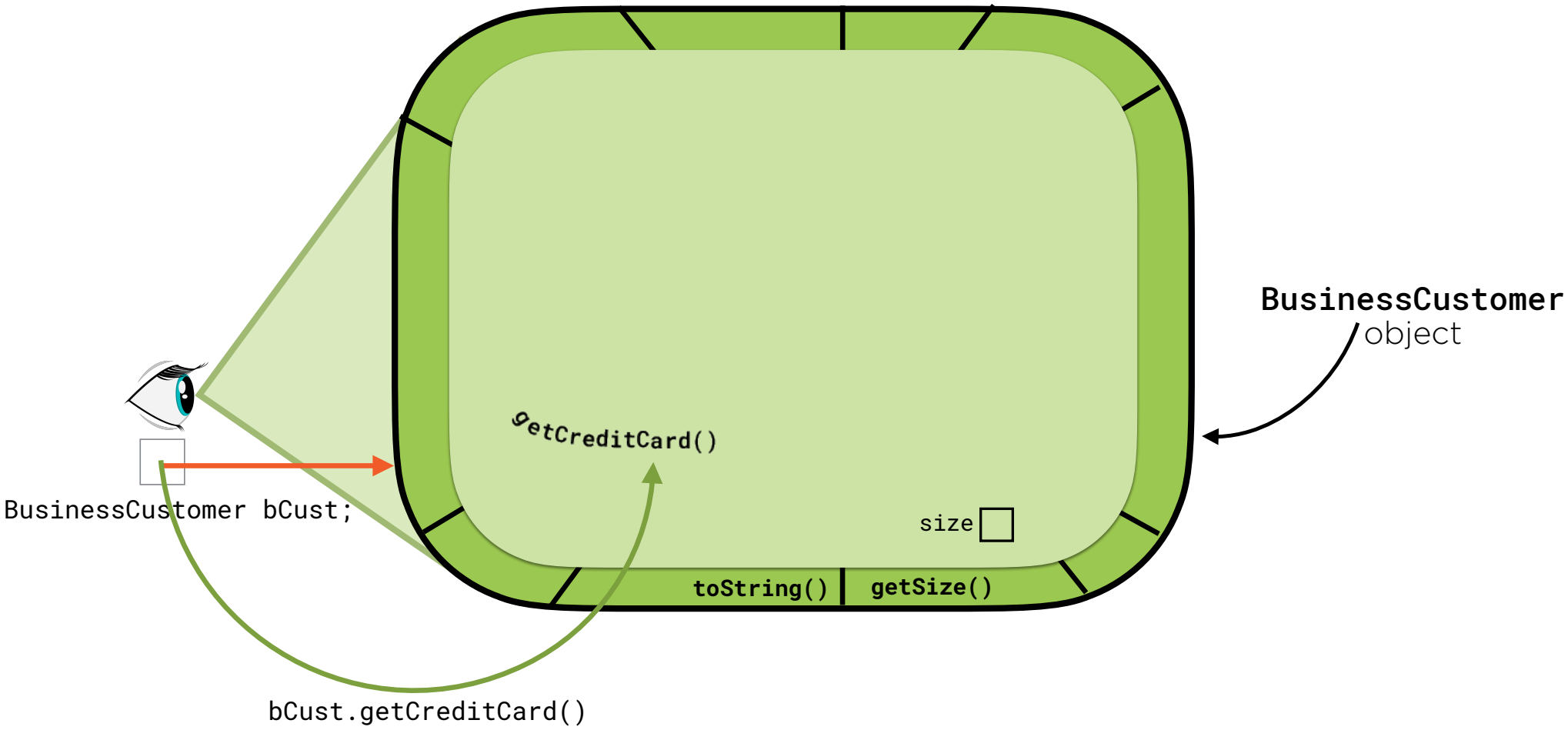
# Understanding Inheritance



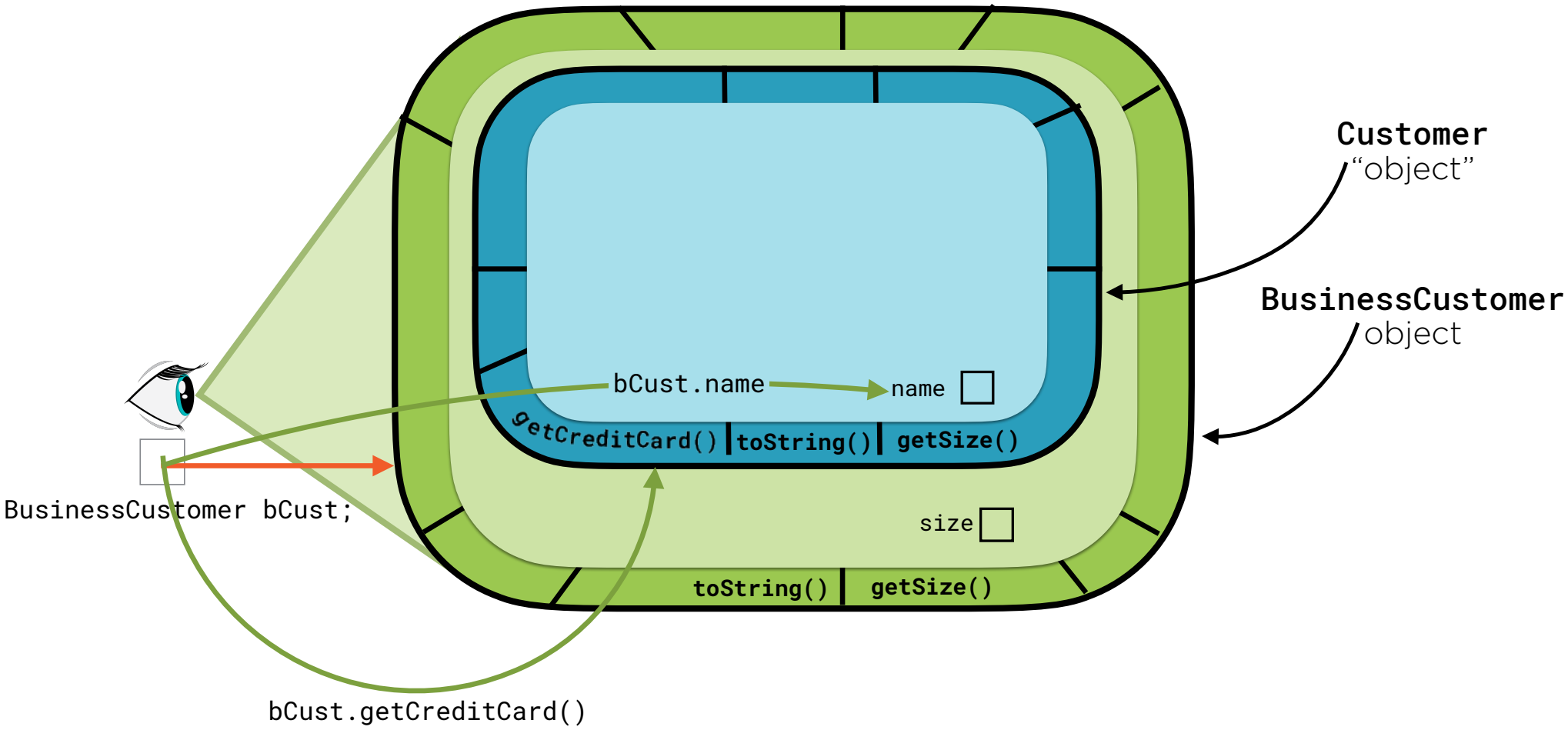
# Understanding Inheritance



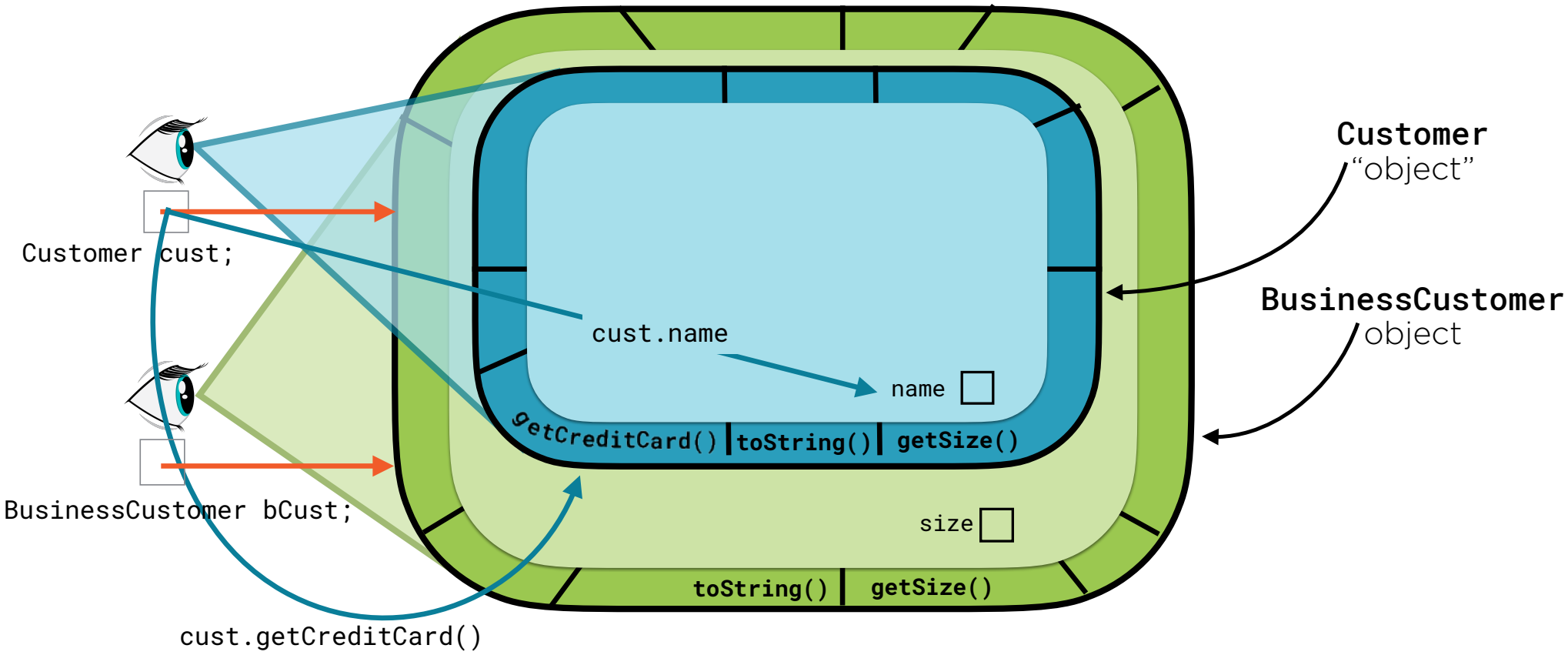
# Understanding Inheritance



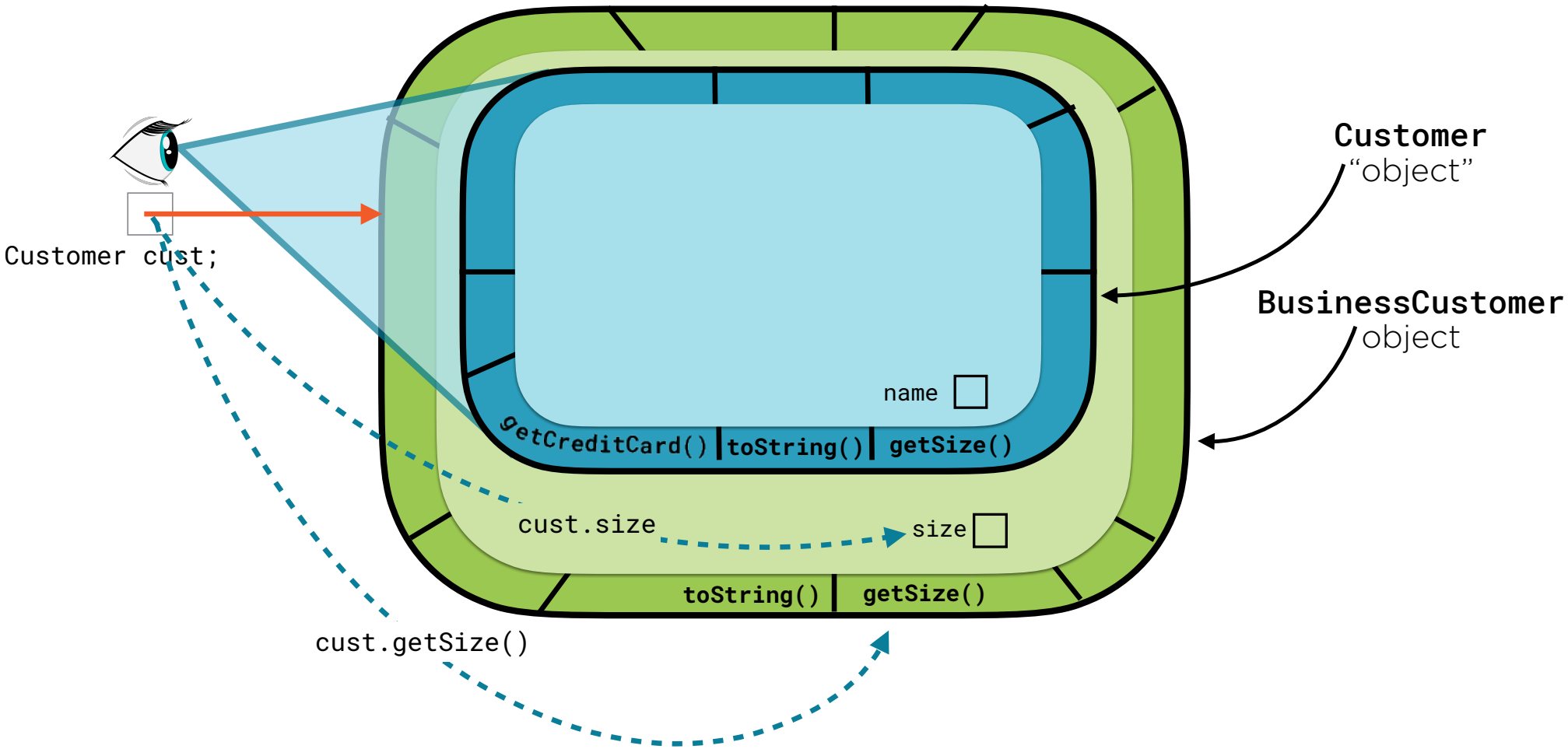
# Understanding Inheritance



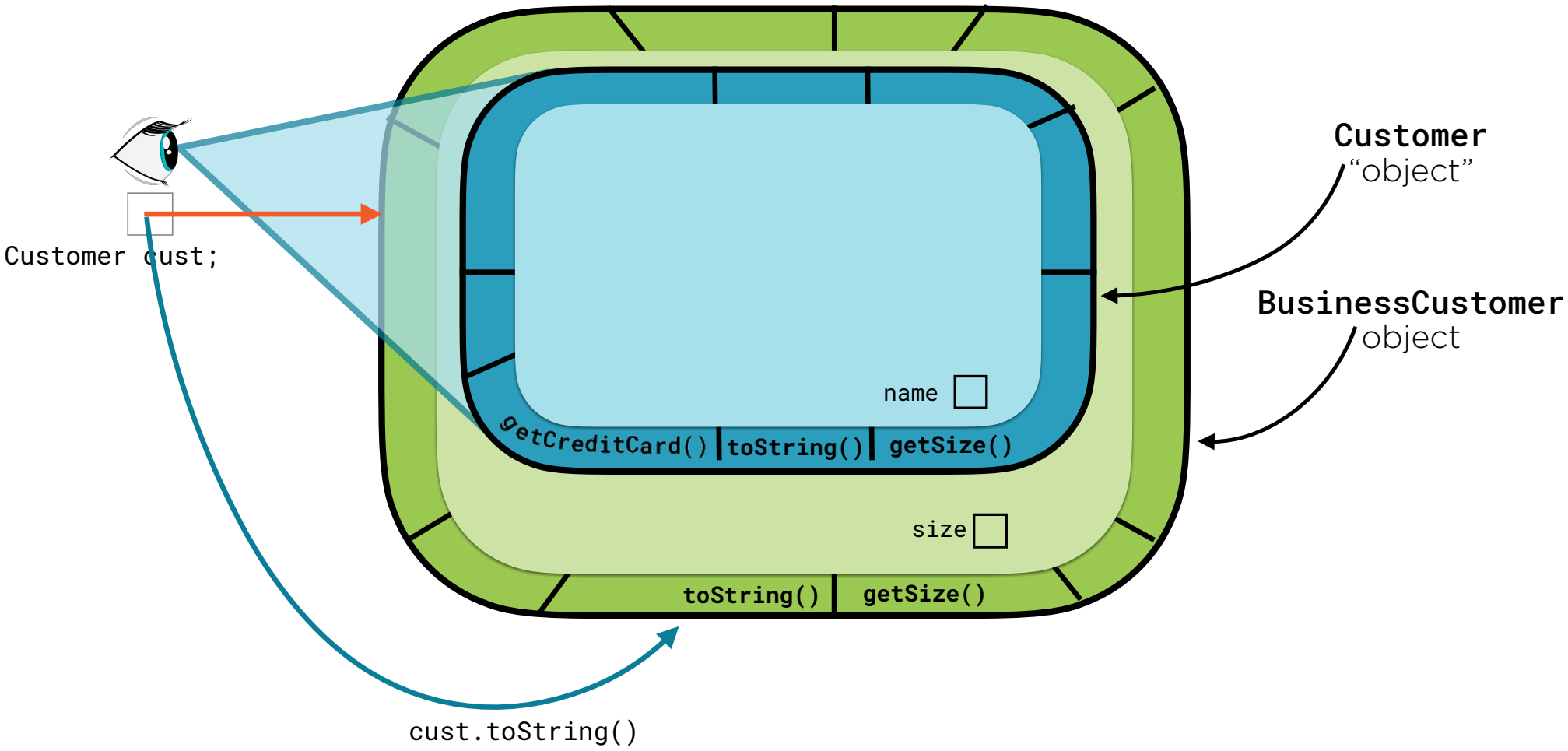
# Understanding Inheritance



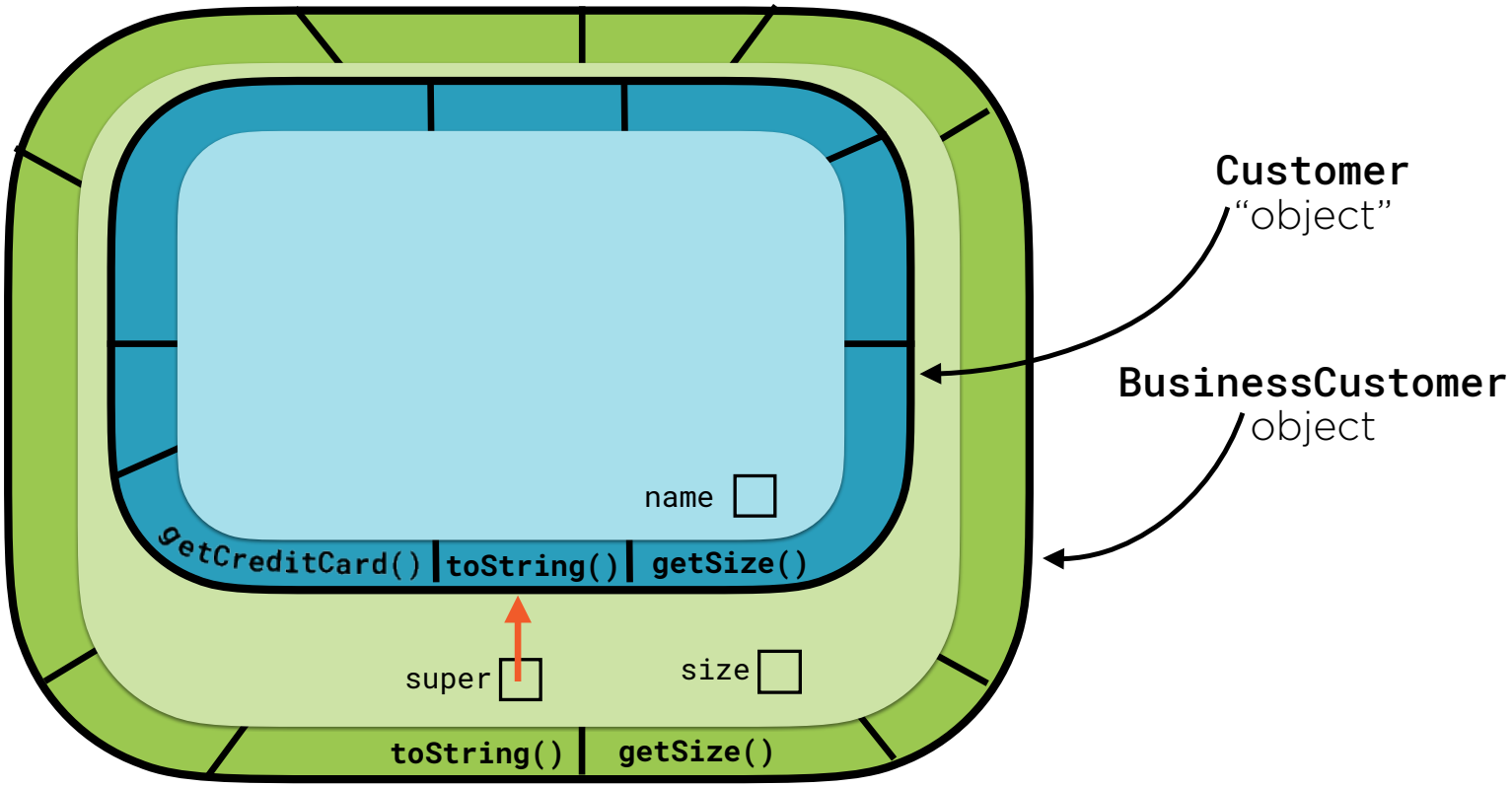
# Understanding Inheritance



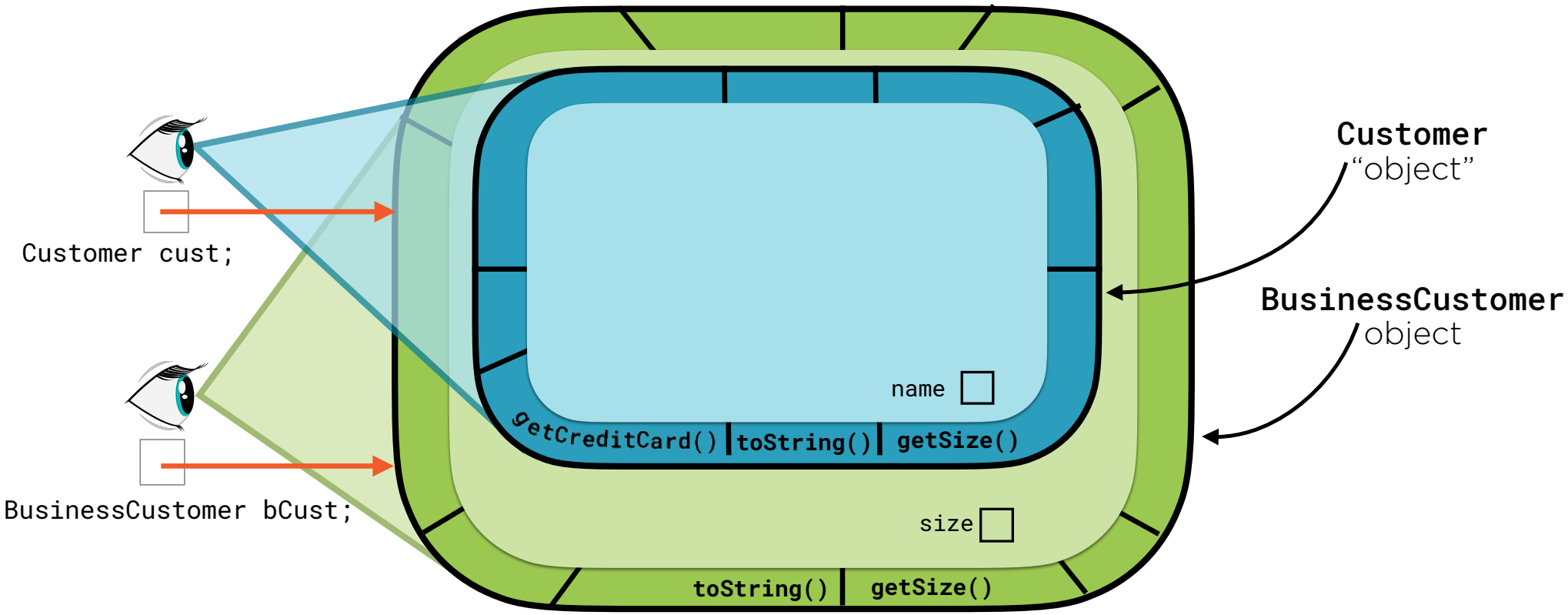
# Understanding Inheritance



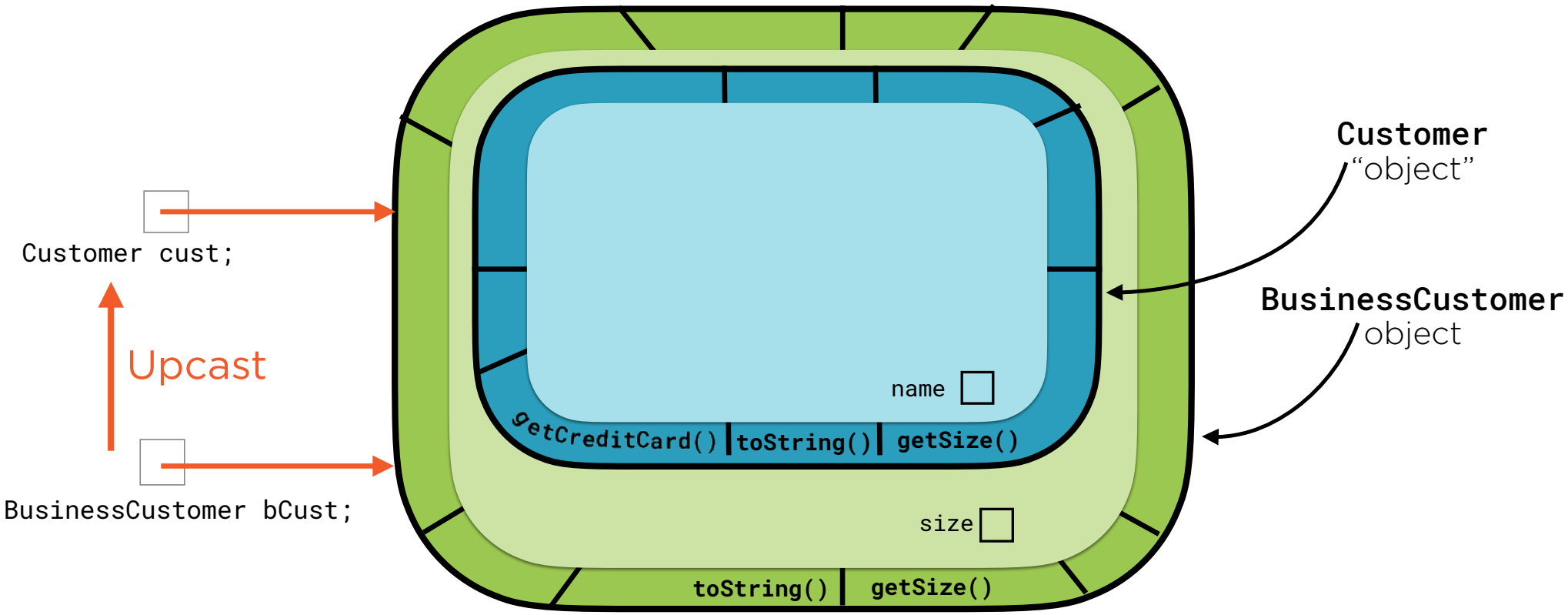
# Understanding Inheritance – **super**



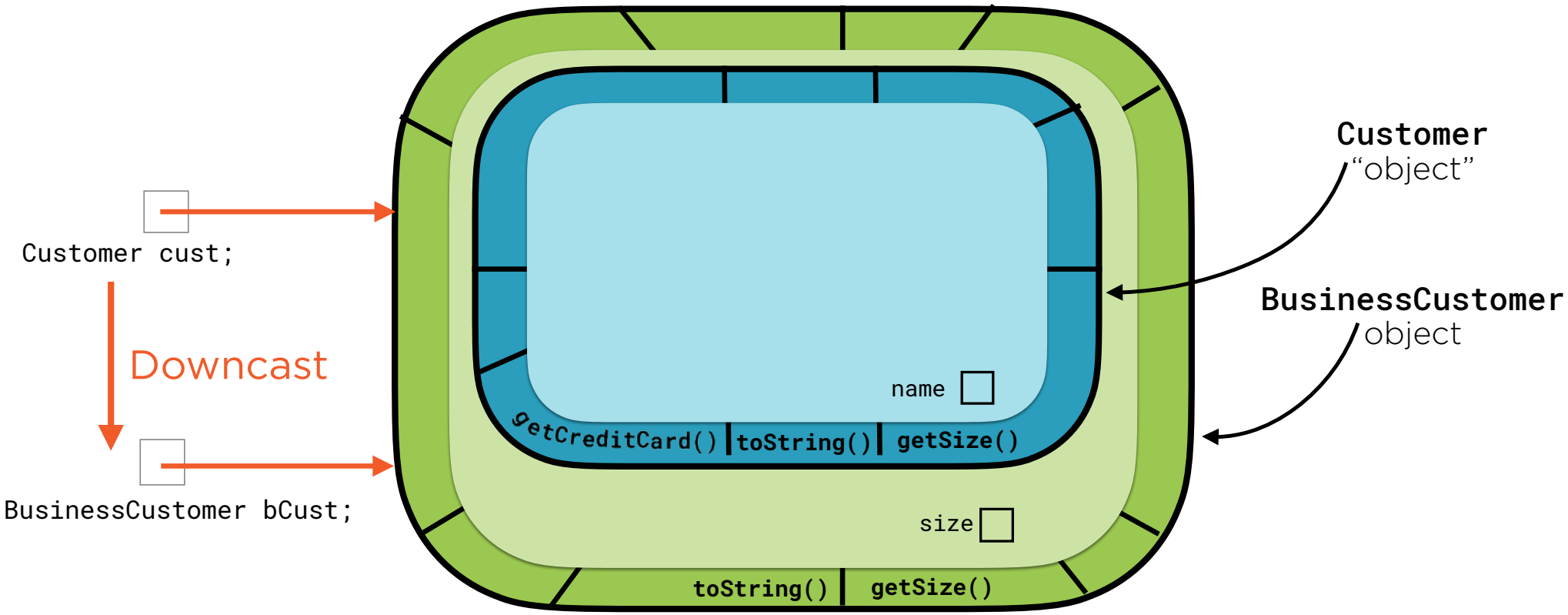
# Understanding Inheritance – Typecasting



# Understanding Inheritance – Typecasting

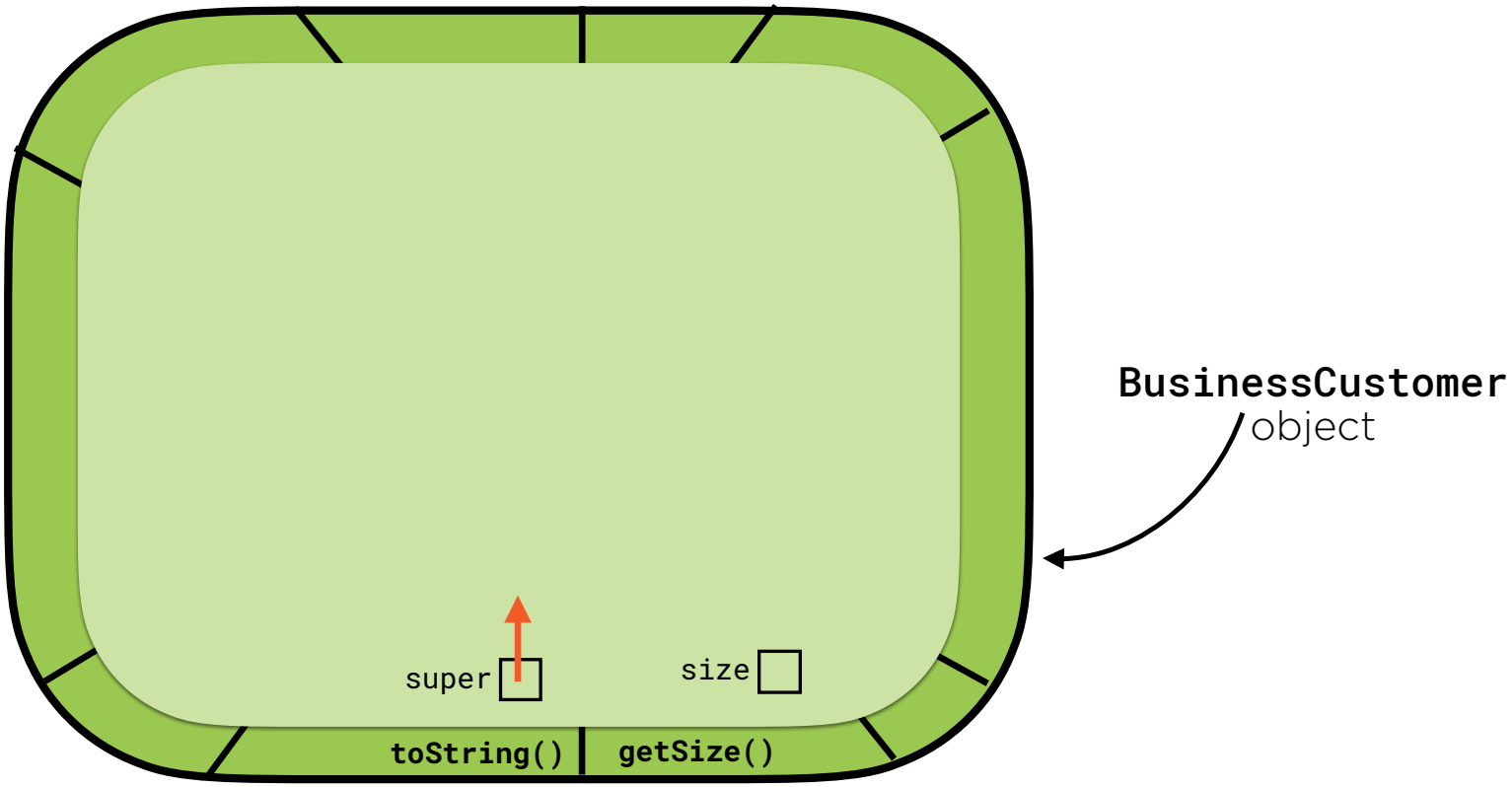


# Understanding Inheritance – Typecasting

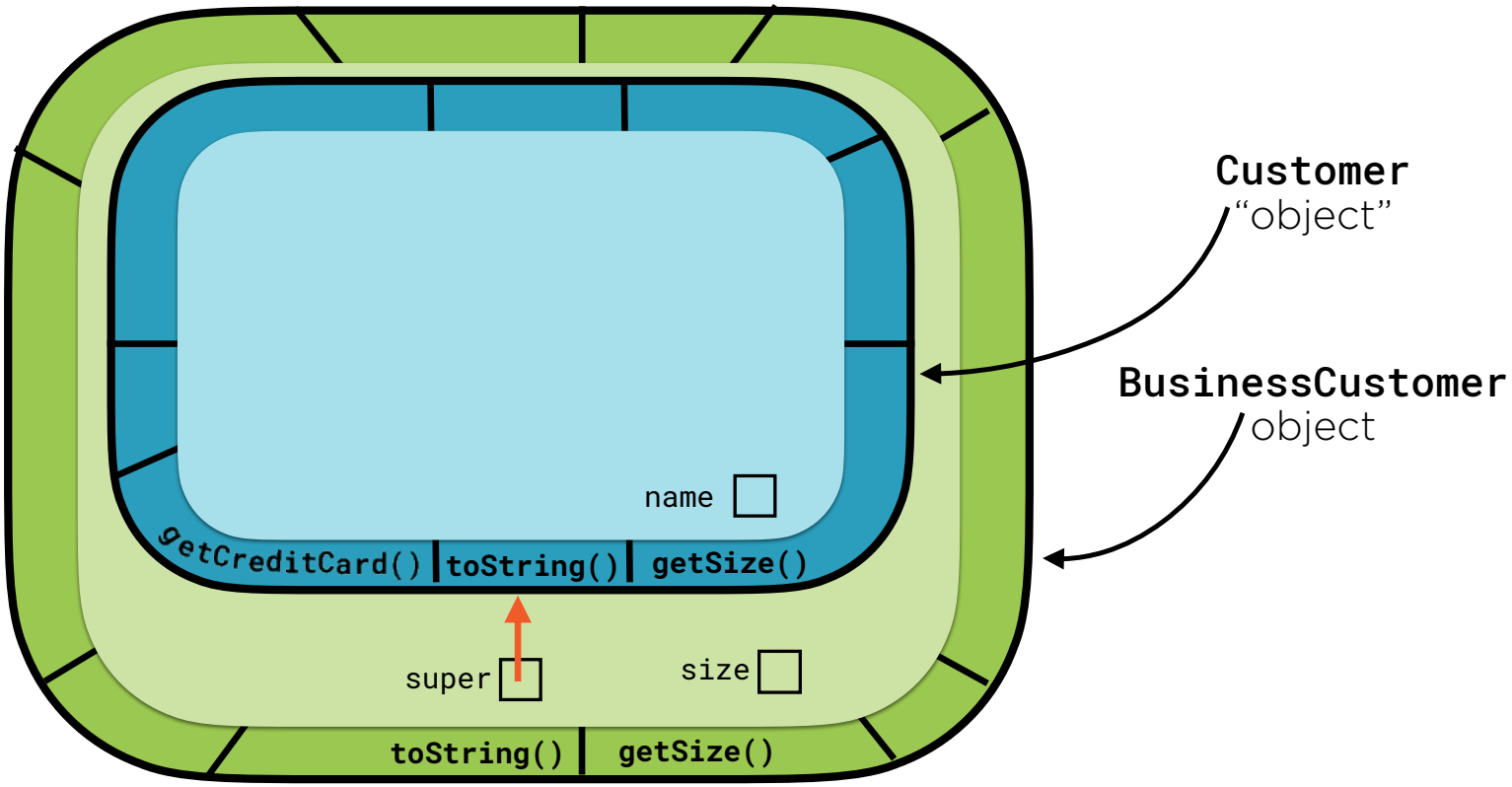


# Demo: Casting

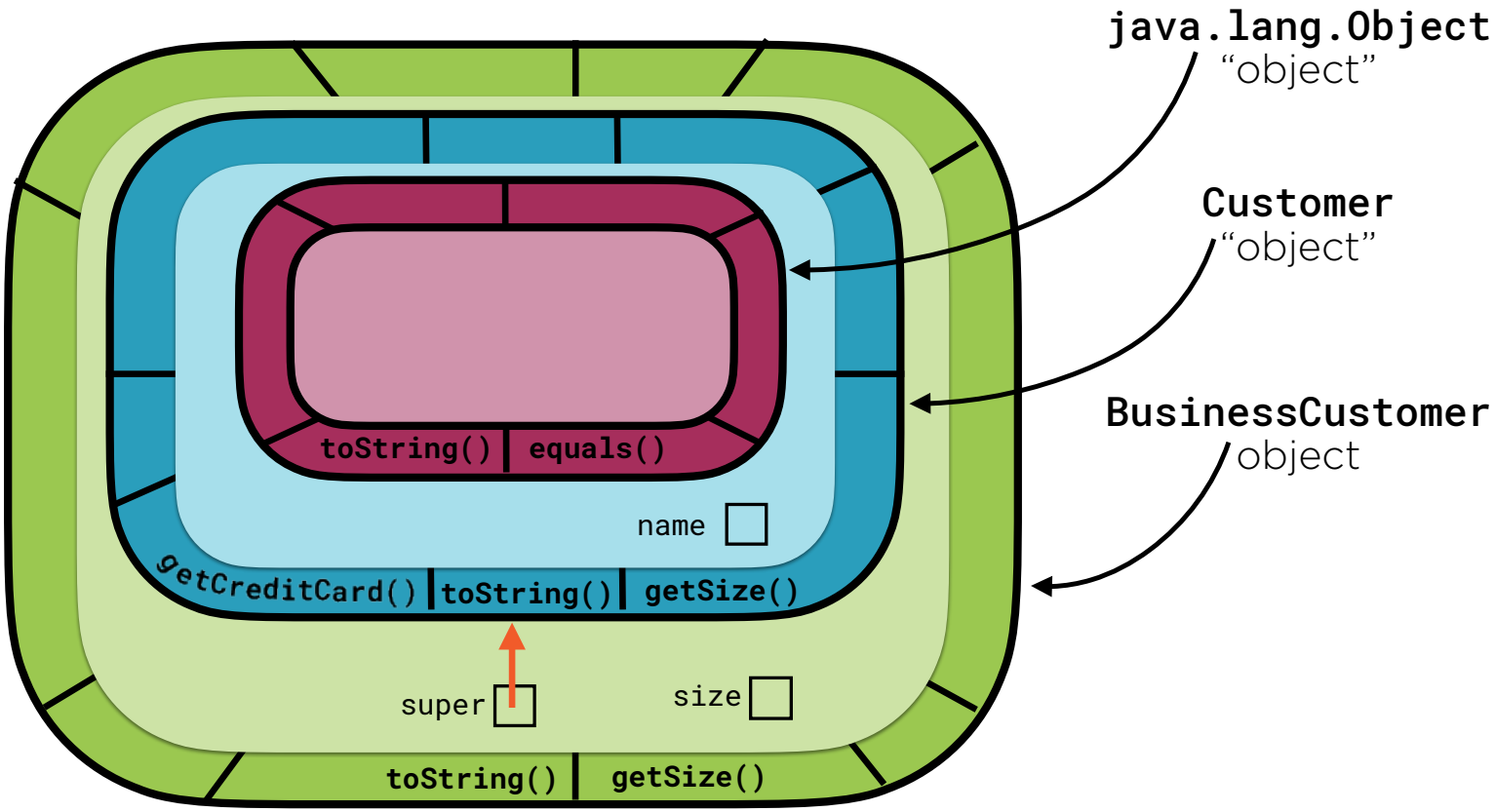
# Object Construction



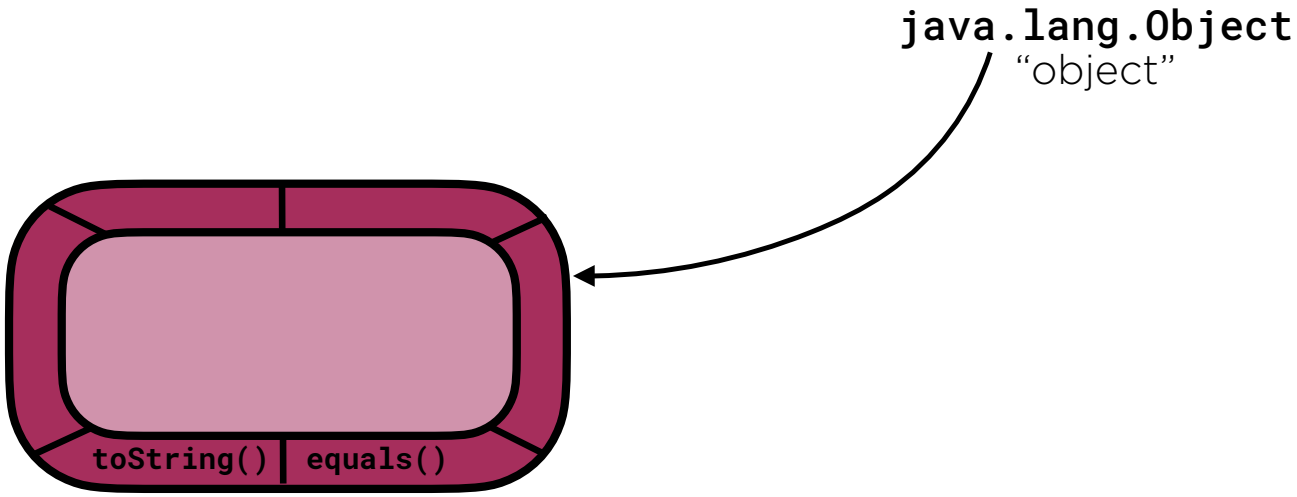
# Object Construction



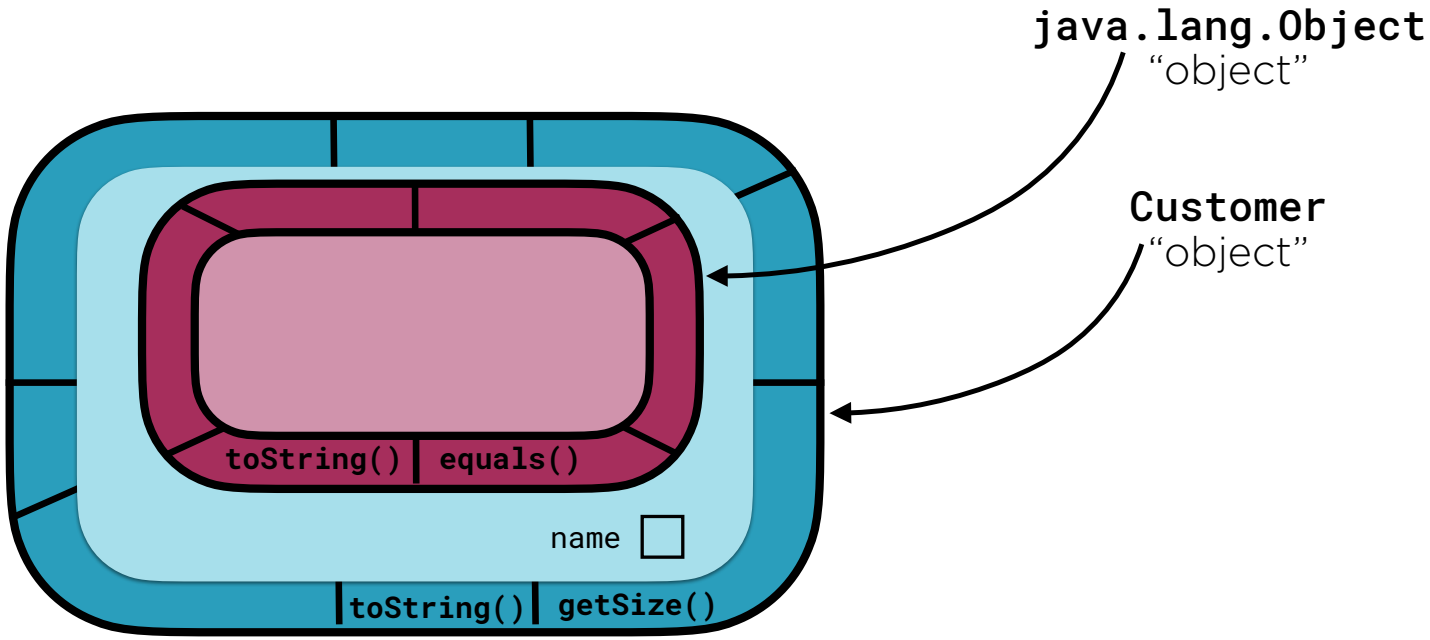
# Object Construction



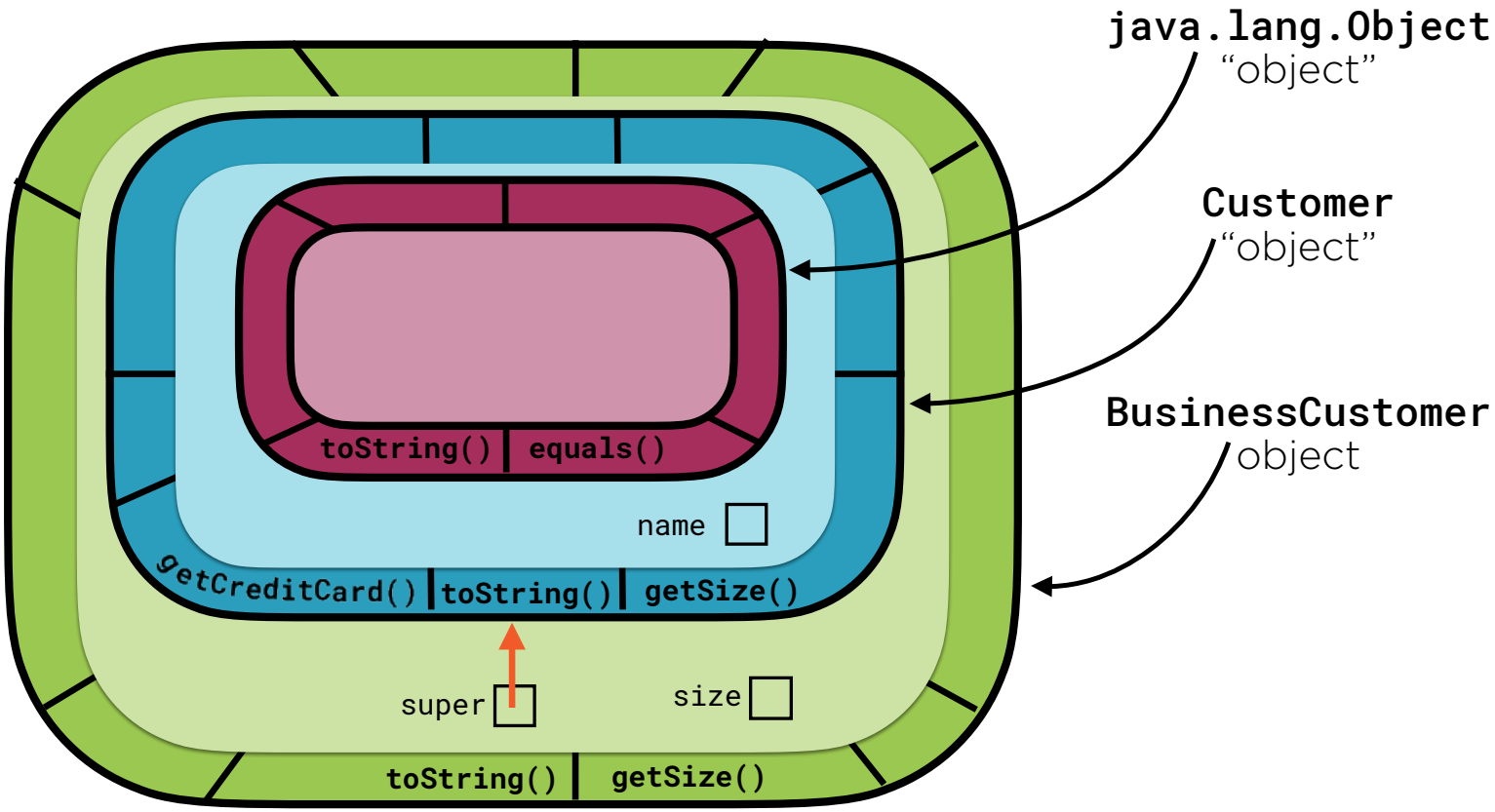
# Object Construction



# Object Construction

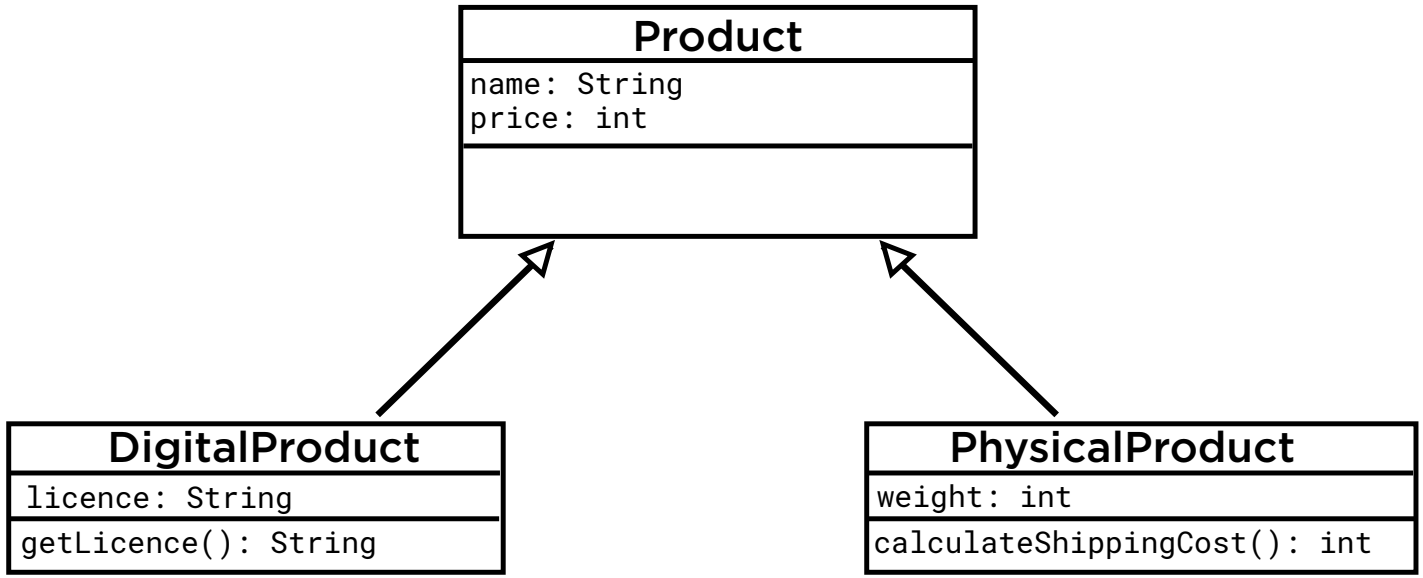


# Object Construction

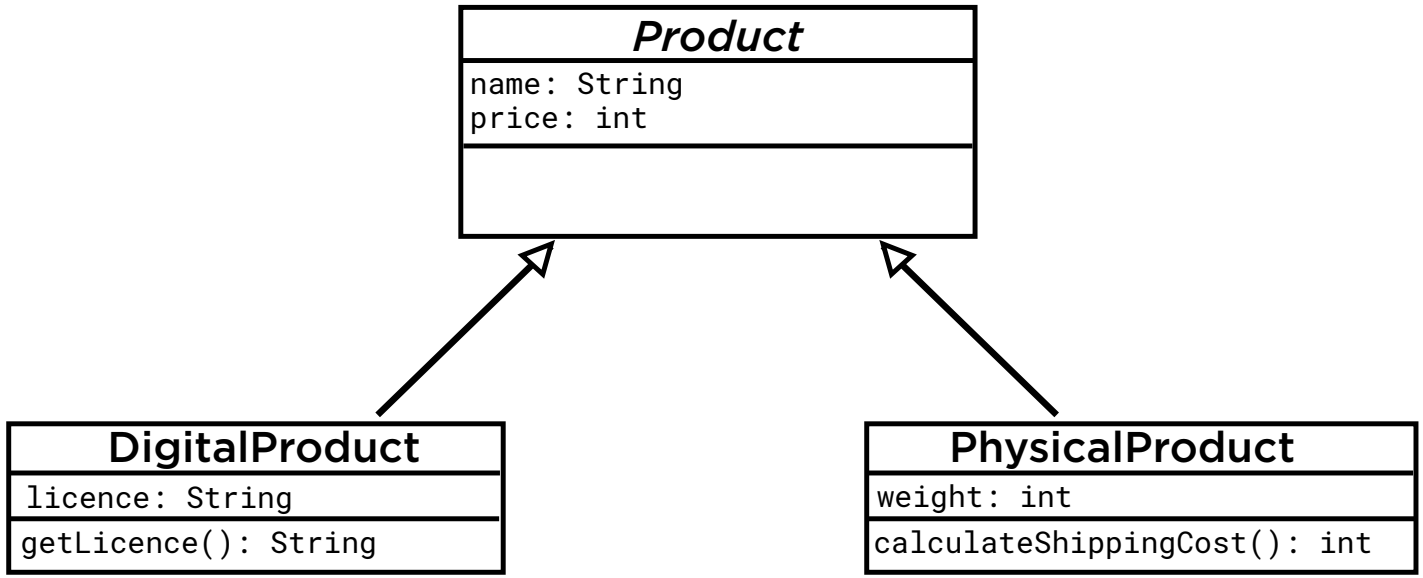


# Aggregation or Inheritance?

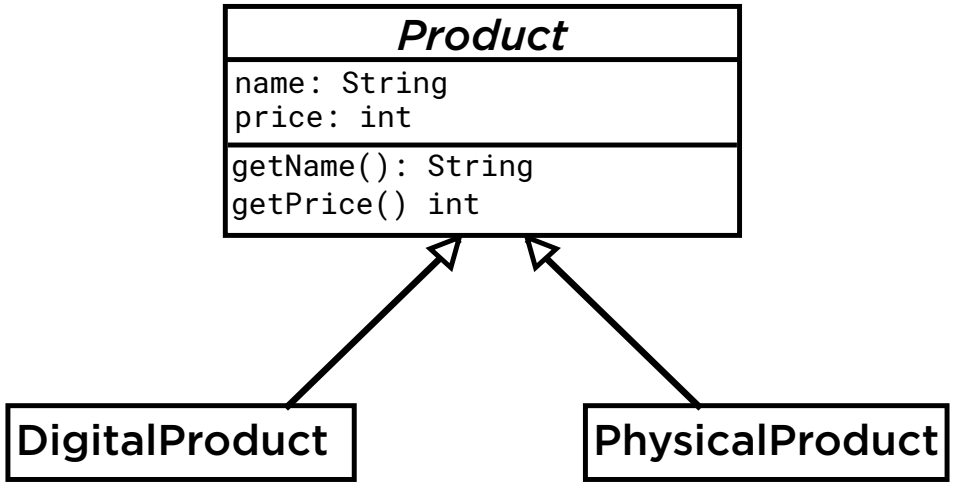
# Using an Abstract Class



# Using an Abstract Class



```java
public abstract class Product {
    private String name;
    private int price;

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        ...
    }

    ...
}
```



# Open–Closed Principle:

Classes and methods should be open for extension, but closed for modification

# The SOLID Principles of Object-Oriented Design

**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**

**D**

# Module Summary

Why Inheritance?

Encapsulating Variation by Subclassing

The Liskov Substitution Principle

Overriding and Dynamic Dispatch

Accessing Overridden Methods

Understanding Inheritance

Abstract Classes

The Open-Closed Principle

**Up Next:**

Interfaces, Composition, and System Design