# LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner - OOP

# COURSE AGENDA

- What is an Object
- Objects
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces

- Collections
- Functional Programming
- Stream
- Exceptions
- Generics
- Arrays

# TOPIC AGENDA

- What is an OOP
- Examples of each Idea
- Does it All Matter?
- What is a Class?
- Inheritance?
- Interfaces
- Summary

# WHAT IS OOP?

- Object-oriented programming (OOP) consists of:
  - **Objects**—Instances of Classes (more later)
  - **Data Abstraction/Encapsulation**—Promotes information hiding and modularity.
  - **Polymorphism**—The same message sent to different objects results in behavior dependent on the object receiving that message.
  - **Inheritance**—Creating new classes and behavior based on existing classes, obtaining code reuse and code structure.
  - **Dynamic binding**—Ability to send messages to objects without knowing their specific type.

# OOP —CLASS EXAMPLES

- **Objects**—Instances of Classes (i.e., Types)
  - Canine
    - Rover
  - University
    - New York University
  - Automobile
    - Ford F-150 (Tag: LUV TRKS)
  - Personal Computer
    - Dell Laptop 1551 (ID Tag: X123 FCV)

# OOP —ABSTRACTION EXAMPLES

- **Data Abstraction/Encapsulation**—Promotes information hiding and modularity.
  - `Print(Document d, Printer p)`
    - Somehow, print the document to the specified printer
  - `Log(String message)`
    - Display the string message to some console, file, or specified stream.

# OOP —POLYMORPHISM EXAMPLES

- **Polymorphism**—The same message sent to different objects results in behavior dependent on the object receiving that message.
  - `Speak(String message)`
    - While a Dog barks, a Person talks, and a Bird Chirps, they all Speak, given a message

# OOP —INHERITANCE EXAMPLES

- **Inheritance**—Creating new classes and behavior based on existing classes, obtaining code reuse and code structure.
- **Animal->Mammal->Canine**
  - Each category has its functions and properties
  - *What about a Platypus?*

# OOP —DYNAMIC BINDING

- **Dynamic binding**—Ability to send messages to objects without knowing their specific type.
  - One of the most important features of OOP
  - Requires a well-thought-out design
  - `Print(Document d, Printer p)`
    - not
  - `Print(PDFDocument doc, ColorPrinter p)`
  - `Print(Invoice inv, LinePrinter p)`

** The secret is Interfaces (more on that later)*

# DOES OOP MATTER?

- Early in OOP, the principles were critical when teaching and building frameworks.
- In practice, well-known frameworks are OOP supporters.
- Most day-to-day programmers thrive without a deep understanding.
  - Are interfaces all we need?
    - Is inheritance overhyped?
  - Data hiding is still critical (it is not a unique principle of OOP)
  - Does Spring Boot and .NET have all the OOP we need?

# WHAT IS A CLASS?

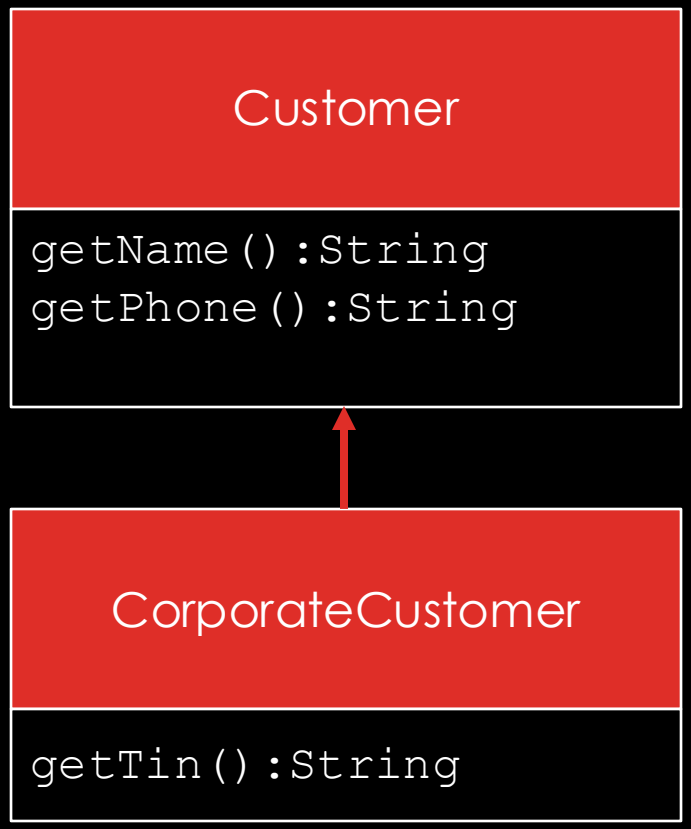
| ShoppingCart |
| --- |
| `Add(Item i)`<br>`Remove(Item i)`<br>`RemoveAll()` |

Class Name

The interface (i.e., programmer's contract) contains all the **public** functions

- How is the `Add(Item i)` implemented?
  - Do you care?
- Why use an `Item` as the type instead of a more specific class?
  - Is `Item` an interface?
- Do you need inheritance?
  - Do you need it now?

# WHAT IS INHERITANCE?




The interface consists of all the functions for a Customer class.

The `CoporateCustomer` Interface consists of all the functions for a Customer **AND** its functions for a total of four functions. The "Is a" relationship.

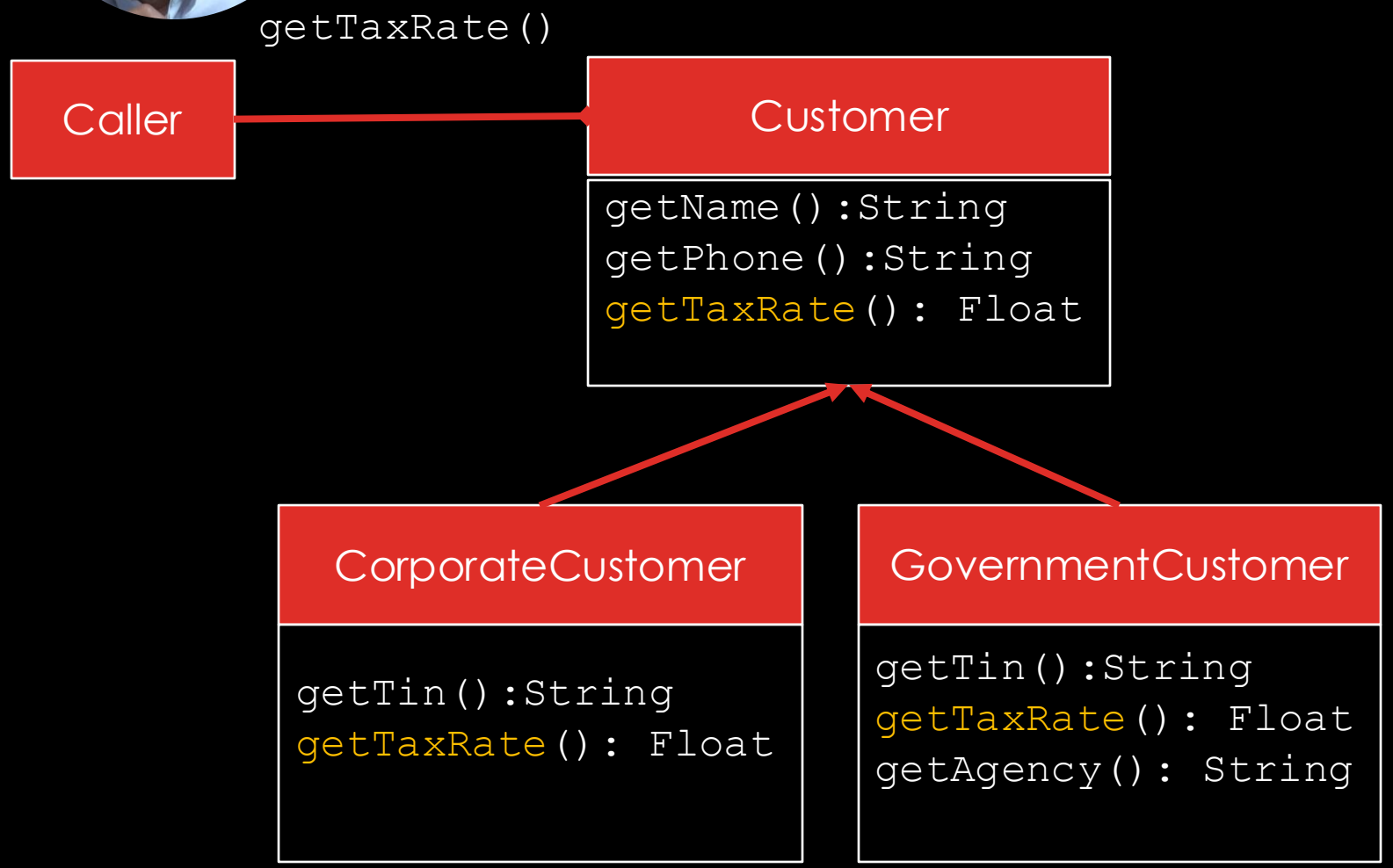
What if the `Customer` class declares a `getTaxRate`?

# WHAT IS COMPOSITION?


The `CoporateCustomer` "has a" relationship to a `ShippingAddress`. The `ShippingAdress` is a class on its own. This is by far a much more common design .

Why do this? Why not just have the address information in the `CoporateCustomer` class?

# WHAT IS POLYMORPHISM?



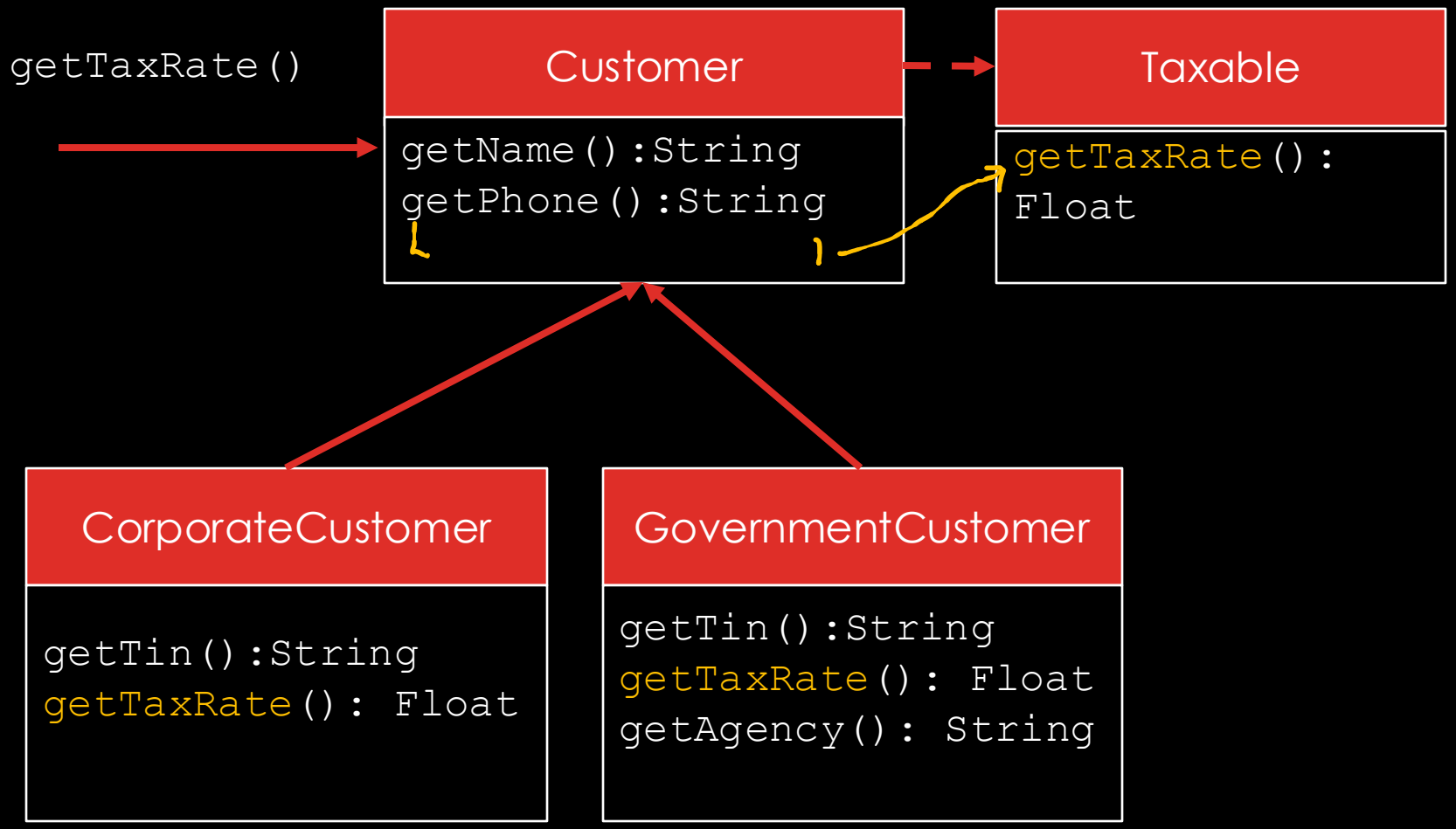

What if we want all customers to be assigned a tax rate? What if we declare a new function in the `Customer` and then expect the children to implement how to calculate that rate?

This is by far a much more standard design.

Why define the function in the `Customer` class? And not in each class independently?

When would you want to do that?

# USING AN INTERFACE

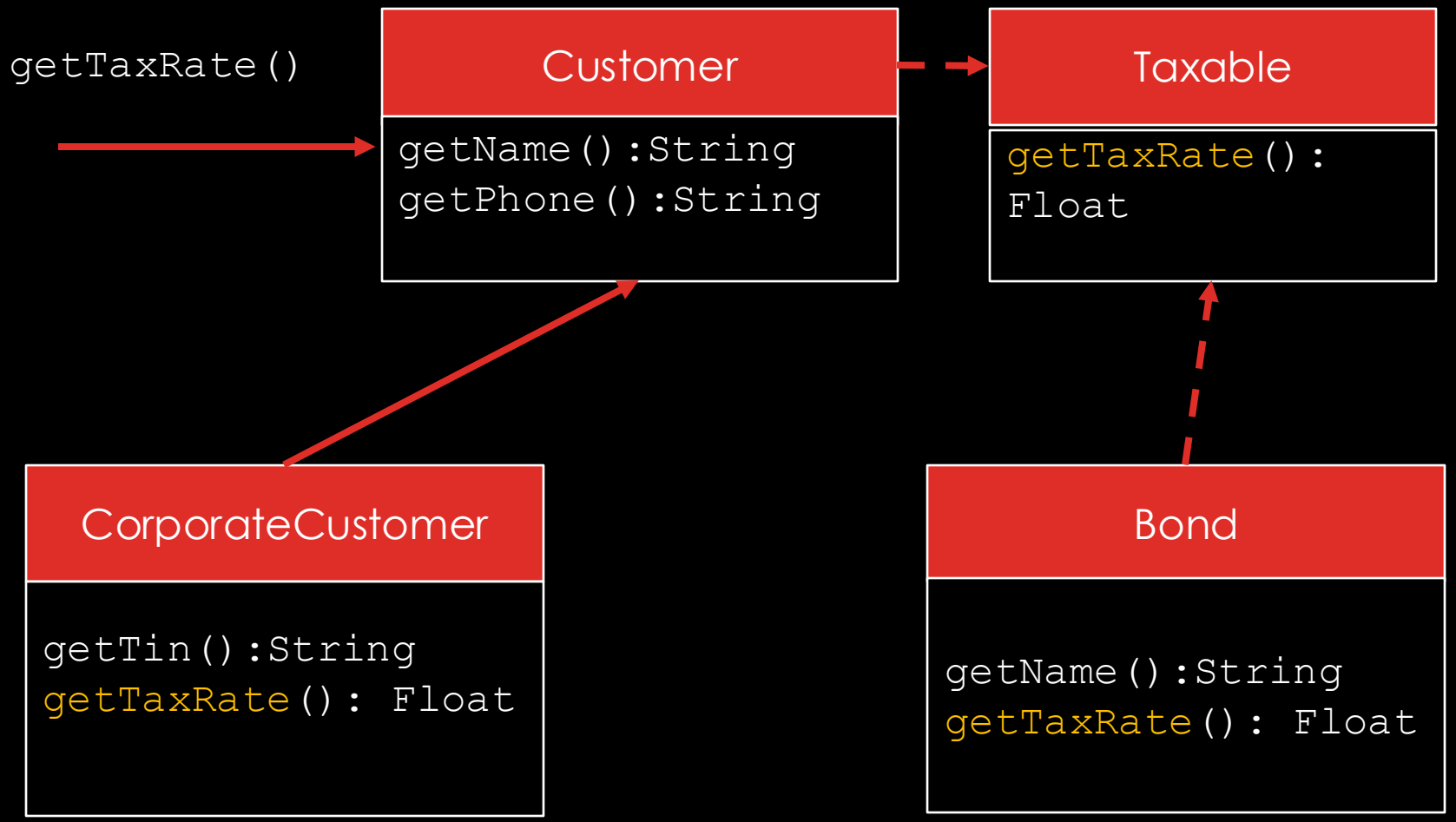



What if, instead, some other classes are taxable? Not just Customers? What if there is no implementation and just an idea that must be implemented?

This is by far a much more common design in OOP.

We create a `Taxable` interface with one method, and then anything " taxable " must be implemented. Correspondingly, any caller is guaranteed a tax rate.

# USING AN INTERFACE




What if, instead, some other classes are taxable? Not just Customers? What if there is no implementation and just an idea that must be implemented?

This is by far a much more common design in OOP.

We create a `Taxable` interface with one method, and then anything " taxable " must be implemented. Correspondingly, any caller is guaranteed a tax rate.

# USING AN INTERFACE V. INHERITANCE

**Inheritance**

Inheritance is used when you have a "`is-a`" relationship, meaning that your subclass is a specific type of the parent class.

Use inheritance when:

- You want to reuse code from a base class.
- You want to establish a clear hierarchical relationship.
- You want to extend or modify the behavior of a base class.

**Interfaces**

Interfaces define a contract for behavior but do not provide the implementation. They are used when you want to define a standard set of methods that different classes must implement. Use interfaces when:

- You want to enforce a set of methods that must be implemented by any class that adheres to the interface.

- You want to ensure that unrelated classes share a certain behavior.
- You want to achieve polymorphism without requiring a strict class hierarchy.

# TOPIC SUMMARY

**OOP**

- Discussed What is OOP
- Examples of
  - Classes and Objects
  - Abstraction Principle
  - Polymorphism
  - Dynamic Binding
  - Inheritance
  - Interfaces
- The difference between defining relationships as parent-child or "is-a" and defining the relationship as "has-a".

**Interfaces v. Inheritance**

- What is the difference between the two
- When to use each
- Polymorphism with each

- Use inheritance when the classes share common attributes and behavior and are logically related (e.g., parent-child relationship).
- Use interfaces when enforcing specific methods across different classes that don't necessarily fit into a strict hierarchy.
- Prefer composition over inheritance whenever possible, which leads to more flexible designs.