Generalization + inheritance Pt. II

Advanced Software Design

Today’s topics

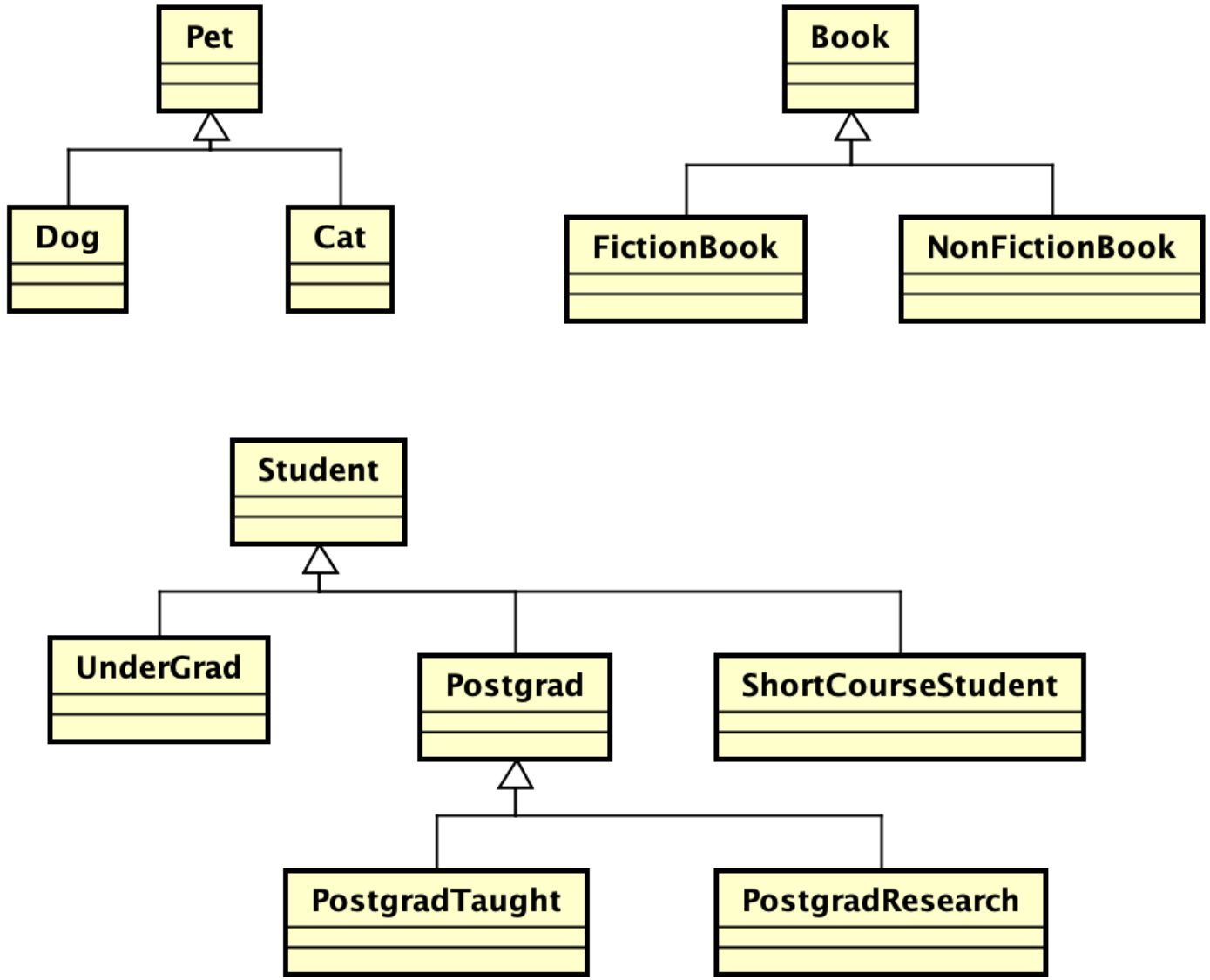
- Revisit topic of generalization
- Discuss inheritance in more detail
- Overriding
- Liskov Substitutability + SOLID
- Interfaces

Generalization

- “Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships. It is a way to construct taxonomic classifications among concepts which are then illustrated in class hierarchies”
  - Craig Larman, “Applying UML and Patterns”, pg. 396, Prentice Hall, 2002

Examples using UML

- Each class hierarchy shows general concepts and relationships to more specialized concepts



Superclass and subclass

- A superclass is a class that is a more general form of something else (also known as base class)
- A subclass is a class that is a more specialized form of something else (also known as derived class)
- What makes a class a more specialized form?
  - It has the same services (behaviour) as the superclass but carries them out differently
  - It has connections to classes that are different to the superclass

Specialisation example

- You open a bank account with £100 on 1st January 2025. On 1st January 2026 (one year later), after making no transactions on the account, you ask for the balance. The balance is £100.

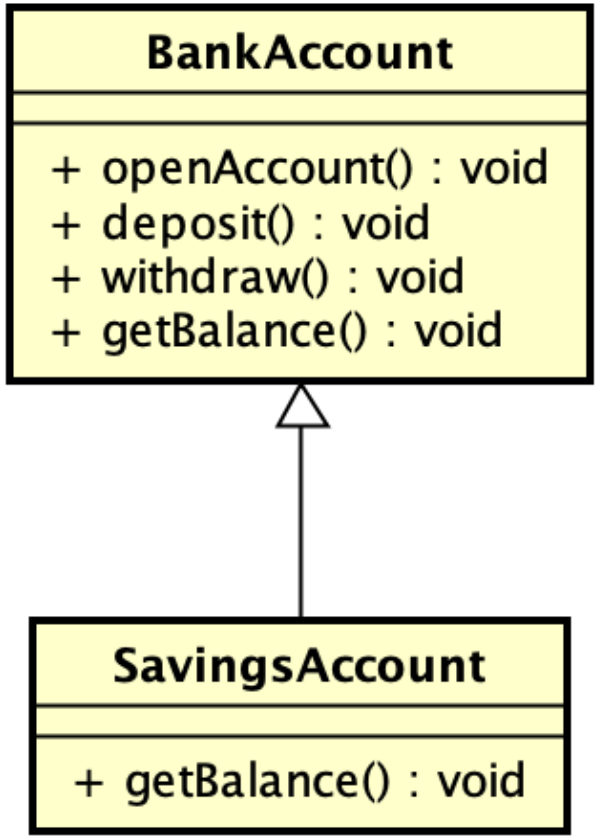
01/01/2025

01/01/2026

- You open a savings account with 10% APR (annual percentage rate) at the same bank with £100 on 1st January 2025. On 1st January 2026 (one year later), after making no transactions on the account, you ask for the balance. The balance is £110.
- Same behaviour…
  - Open account, deposit money, get balance
- … but handled in a different way

Superclass - subclass

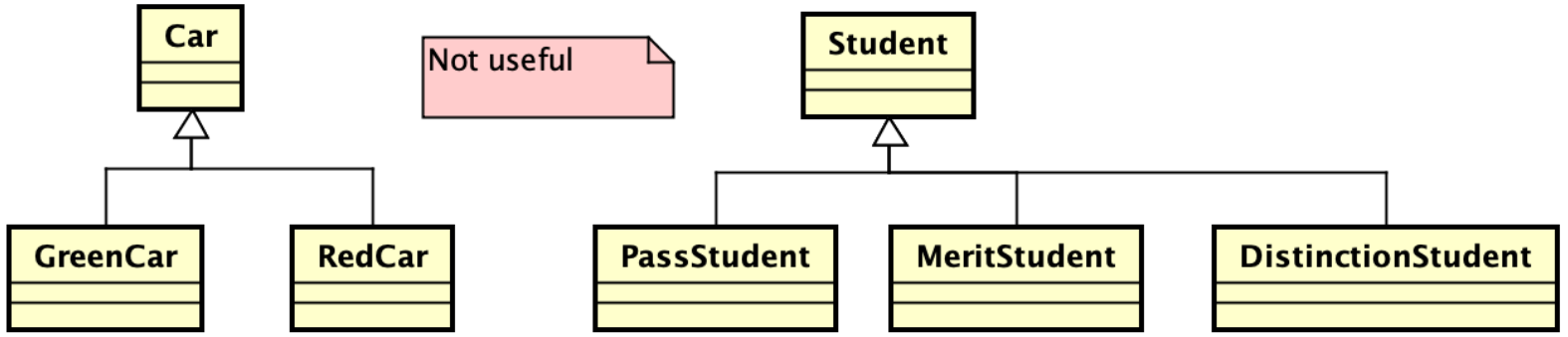
- We can express the fact that the savings account has the same behaviour as the bank account but handled in a different way using generalization
- BankAccount is the more general concept
- SavingsAccount is a specialization of the more general concept BankAccount

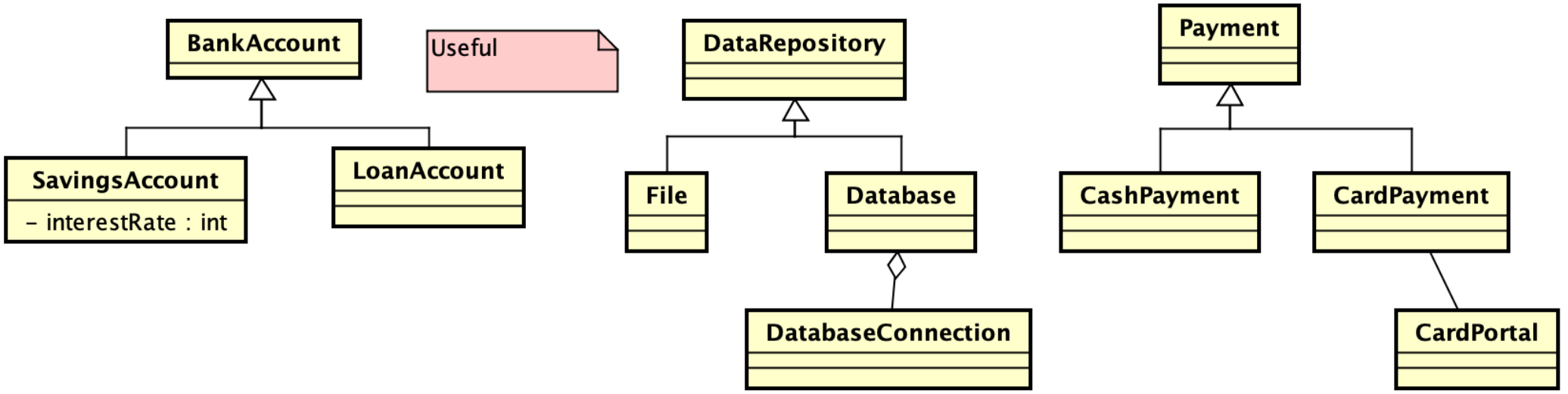


When to create subclasses?

- When is it useful to use generalization and create subclasses of a class?
- “Applying UML and Patterns”, pg. 401
  - When subclass has a different set of attributes – e.g, SavingsAccount will have an additional attribute to store interest rate
  - Subclass has different class relationships - e.g, associations/aggregations/compositions with other classes
  - Subclass does the same things as the superclass but in a materially different way

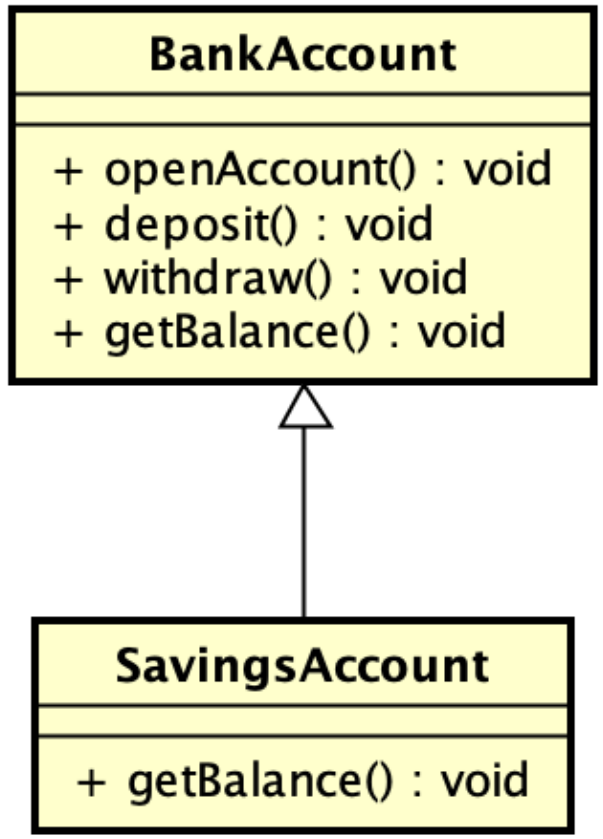
Examples


Car
Not useful
Student
GreenCar
RedCar
PassStudent
MeritStudent
DistinctionStudent


BankAccount
Useful
DataRepository
Payment
SavingsAccount
– interestRate : int
LoanAccount
File
Database
CashPayment
CardPayment
DatabaseConnection
CardPortal

Is-A Rule

- Any object created from a subclass is an object of the superclass
- IS-A rule
- ”A savings account IS-A bank account”
- Why? Because the generalization relationship says that SavingsAccount is just a specialized form of BankAccount



When to create superclasses

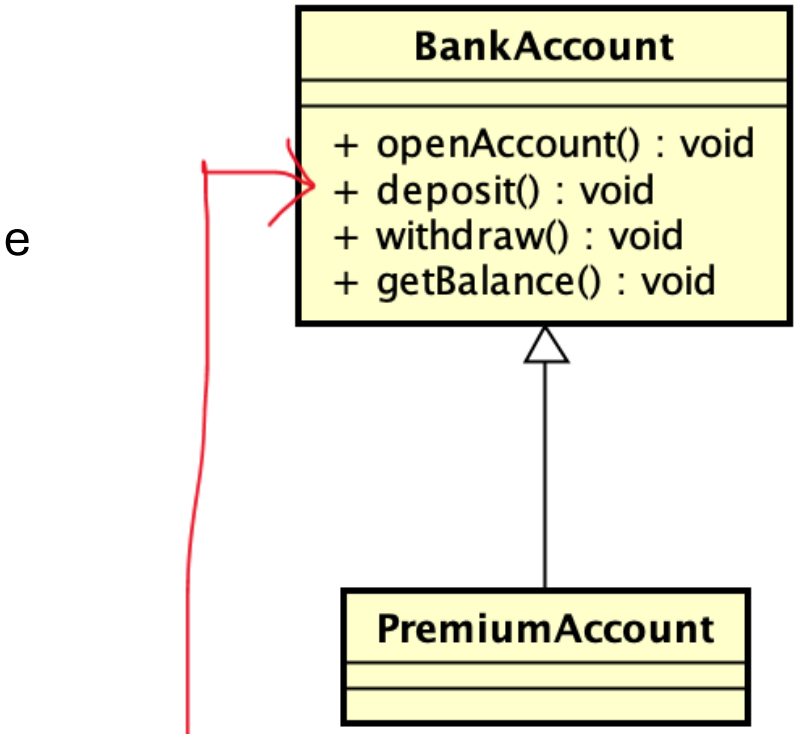
- We assume we have a set of classes and we wonder if we should make them subclasses of some more general concept
- Create a superclass when (“Applying UML and Patterns”, pg. 403):
  - Classes that are potential subclasses represent variations of each other or of a similar concept
  - Classes have same attribute that can be factored out
  - Classes have same behaviour

Inheritance

- Inheritance ”gives” us generalization – it provides the grounds for saying that one thing is a specialized form of another thing
- What do we mean by “inheritance”? Something passed on by one thing to another because of a relationship – e.g., parent-child, predecessor-successor
  - “I inherited my love of gardening from my mum/dad”
- In object-oriented development, inheritance means inheritance of **behaviour** – what is done

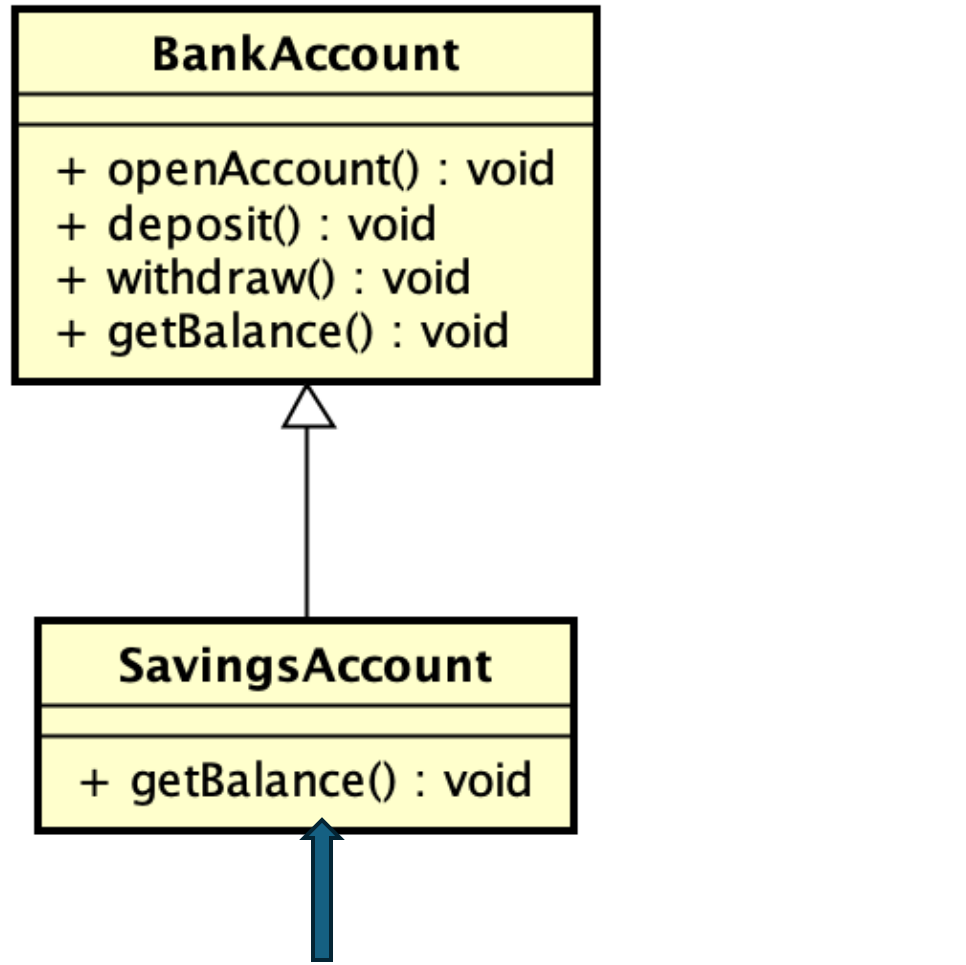
Inheritance of behaviour

- In the BankAccount example, suppose we define a new subclass called PremiumAccount
- Inheritance of behaviour means that the behaviour of the superclass is inherited by the subclass – i.e., a PremiumAccount can:
  - openAccount
  - deposit
  - withdraw
  - getBalance
- In this example class hierarchy, not only the behaviour (what is done) is inherited, but also how it is done (the implementations) is inherited. For example, if we deposit into a PremiumAccount, we are using the BankAccount version of this behaviour



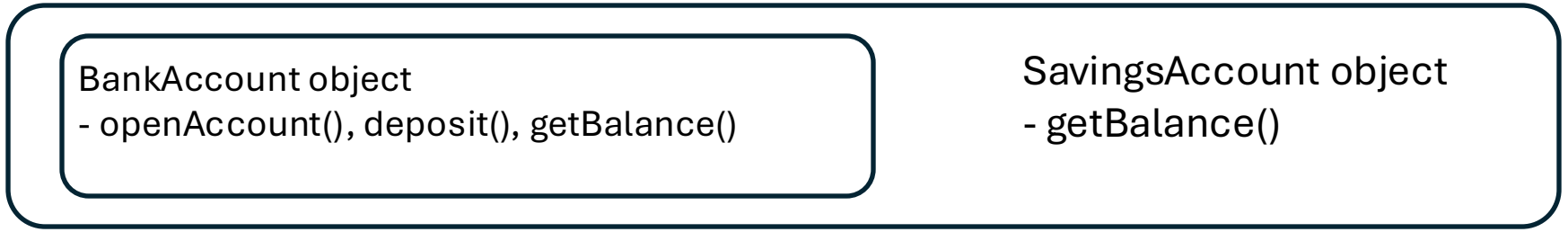
Overriding

- Sometimes we want to inherit behaviour but change how it's done
- For example, we want to inherit all the behaviour of BankAccount (what is done) …
- … but change how it's done - e.g, for getBalance
- We still want to get the balance, but we want to do it in a different way
- In these cases, we want to "override" the inherited behaviour with a new version – as shown in this class diagram
- The subclass will still have the same behaviour as the superclass but will do some things differently



How inheritance is (usually) implemented

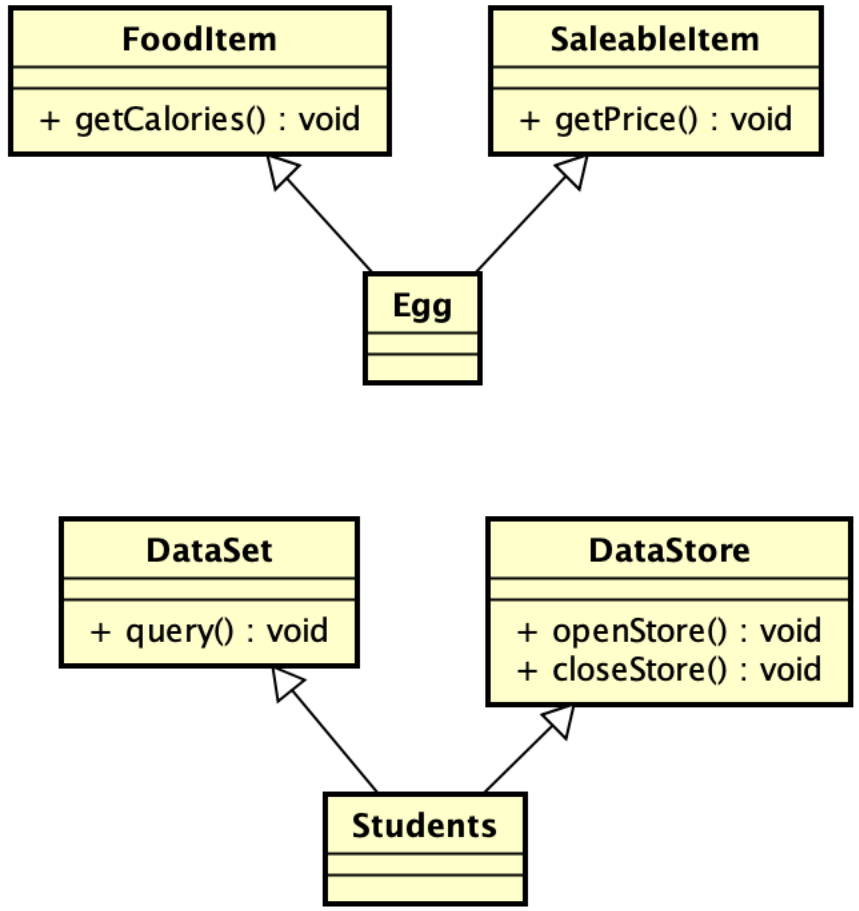
- Inheritance is (usually – caveat!) implemented by (a kind of) **nesting** the superclass object within the subclass object



- E.g., when a SavingsAccount object receives a deposit() message, it first looks in its own methods to find one that matches– if it's not there, then it looks in the nested BankAccount object to find a matching method

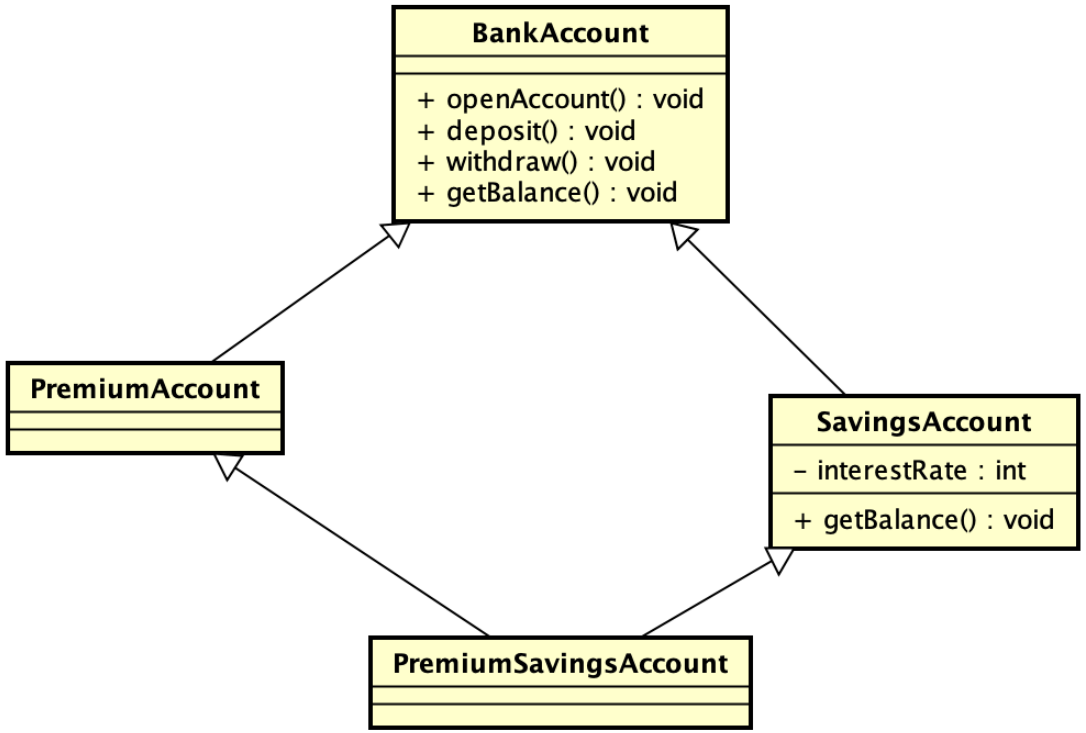
Types of inheritance

- There are two types of inheritance
- Single inheritance – each subclass has only one direct superclass
- Multiple inheritance – a subclass can potentially have more than one superclass
- Multiple inheritance often used to inherit public interface from one class and private interface (supporting functions) from another – e.g, the 2nd example
- Note – Java does NOT support multiple inheritance BUT C++ does
- https://javapapers.com/core-java/why-multiple-inheritance-is-not-supported-in-java/

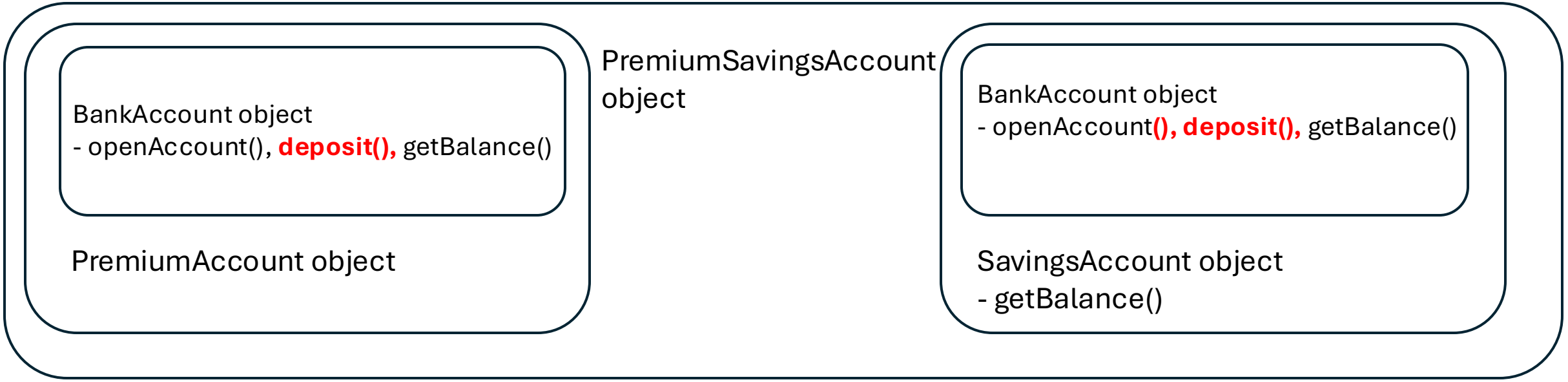


The Diamond problem of multiple inheritance

- Multiple inheritance can be useful (in rare cases), but it has a problem, sometimes called the “diamond problem”
- If a PremiumSavingsAccount object receives a deposit() message, what happens?



The diamond problem explained (hopefully)



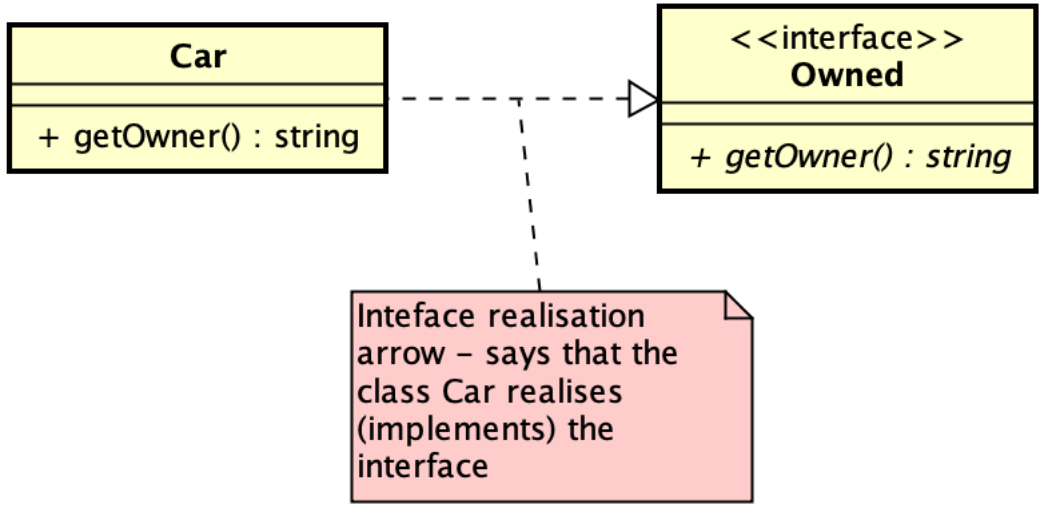
Assume we send the deposit() message to this PremiumSavingsAccount object – it has TWO deposit messages in it that match – which does it use?

Interfaces

- Java uses concept of interfaces to allow the same kind of thing that multiple inheritance provides – getting behaviour in a class from more than one source – without the diamond problem
- An interface is simply a specification of behaviour – what is done – that doesn't say how it is done
  - It simply lists the names of methods (with return types and parameter types) but no code
- A class "realizes" an interface if it has methods (with code) for all the methods named in the interface
- Since we're not "nesting" one object inside another, a class can "realise" many interfaces without suffering the diamond problem

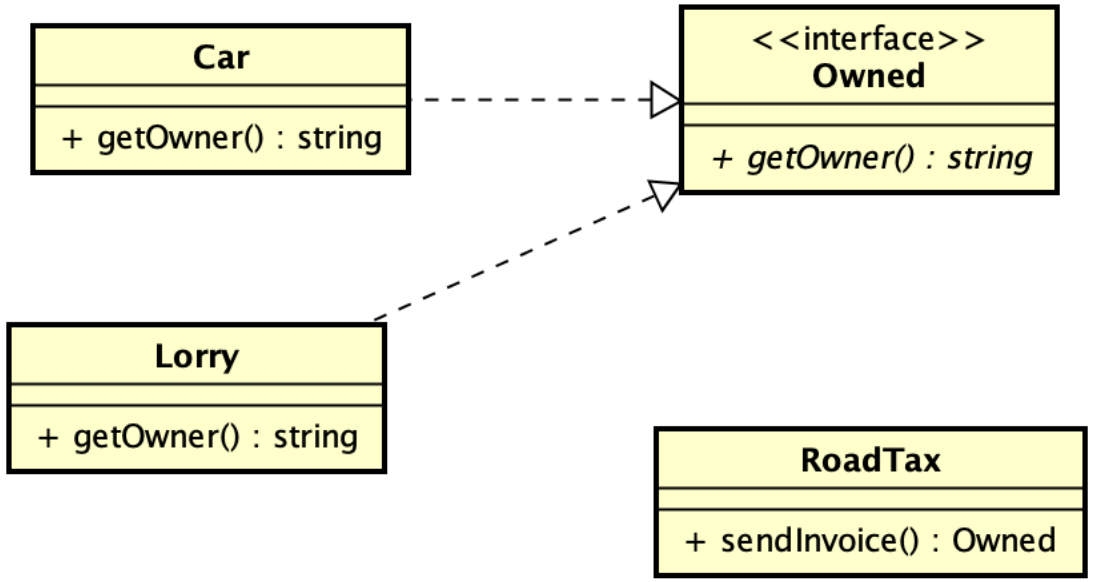
Example

- In this example, we show the interface and the class that realizes it (using an interface realization arrow) – this says that the class has implementations (real methods) for the behaviour specified in the interface



Using interfaces

- Interfaces can be used as types
- When an interface is used as a type, it means that we expect a value of that type has the behaviour specified in the interface
- In this example, Car and Lorry can be used interchangeably as arguments to the sendInvoice() method of RoadTax, even though they don't have a common superclass



Advantages and disadvantages of interfaces

- We can implement many interfaces
- We can create opportunities for polymorphism even in cases where classes don't share a common more general concept
- Both Cheese and Magazine can be added using the same addToBasket method in ShoppingBasket without having a common superclass
- BUT – interfaces don't provide inheritance because there is nothing to inherit